

# Description of the course "Lessons on development of 64-bit C/C++ applications"

The course is devoted to creation of 64-bit applications in C/C++ language and is intended for the Windows developers who use Visual Studio 2005/2008/2010 environment. Developers working with other 64-bit operating systems will learn much interesting as well. The course will consider all the steps of creating a new safe 64-bit application or migrating the existing 32-bit code to a 64-bit system.

The course is composed of 28 lessons devoted to introduction to 64-bit systems, issues of building 64-bit applications, methods of searching errors specific to 64-bit code and code optimization. Such questions are also considered as estimate of the cost of moving to 64-bit systems and rationality of this move.

**The authors of the course:** candidate of physico-mathematical sciences Andrey Nikolaevich Karpov and candidate of technical sciences Evgeniy Alexandrovich Ryzhkov. The authors are involved in maintaining the quality of 64-bit applications and participate in development of PVS-Studio static code analyzer for verifying the code of resource-intensive applications.

**The rightholder of the course** is OOO "Program Verification Systems". The company's site: <http://www.viva64.com>. Contacts: e-mail: [support@viva64.com](mailto:support@viva64.com), 300027, Tula, PO box 1800.

## The contents of the course

- Lesson 01. What 64-bit systems are.
- Lesson 02. Support of 32-bit applications.
- Lesson 03. Porting code to 64-bit systems. The pros and cons.
- Lesson 04. Creating the 64-bit configuration.
- Lesson 05. Building a 64-bit application.
- Lesson 06. Errors in 64-bit code.
- Lesson 07. The issues of detecting 64-bit errors.
- Lesson 08. Static analysis for detecting 64-bit errors.
- Lesson 09. Pattern 01. Magic numbers.
- Lesson 10. Pattern 02. Functions with variable number of arguments.
- Lesson 11. Pattern 03. Shift operations.
- Lesson 12. Pattern 04. Virtual functions.
- Lesson 13. Pattern 05. Address arithmetic.
- Lesson 14. Pattern 06. Changing an array's type.
- Lesson 15. Pattern 07. Pointer packing.
- Lesson 16. Pattern 08. Memsize-types in unions.
- Lesson 17. Pattern 09. Mixed arithmetic.
- Lesson 18. Pattern 10. Storage of integer values in double.
- Lesson 19. Pattern 11. Serialization and data interchange.
- Lesson 20. Pattern 12. Exceptions.
- Lesson 21. Pattern 13. Data alignment.
- Lesson 22. Pattern 14. Overloaded functions.
- Lesson 23. Pattern 15. Growth of structures' sizes.
- Lesson 24. Phantom errors.
- Lesson 25. Working with patterns of 64-bit errors in practice.
- Lesson 26. Optimization of 64-bit programs.
- Lesson 27. Peculiarities of creating installers for a 64-bit environment.

- Lesson 28. Estimating the cost of 64-bit migration of C/C++ applications.

**The course's duration:** the course implies that you study each of the 28 lessons on your own in 20-40 minutes. The total time of studying the material is about 18 hours.

---

Powered by [RSDN Authoring Pack](#)

## Lesson 1. What 64-bit systems are

By the moment of writing the course, there are two most popular [64-bit](#) architectures of microprocessors: IA64 and Intel 64.

1. **IA-64** is a 64-bit microprocessor architecture developed by Intel and Hewlett Packard companies together. It is implemented in Itanium and Itanium 2 microprocessors. To learn more about the architecture IA-64 see the following Wikipedia article "[Itanium](#)".
2. **Intel 64** (EM64T / AMD64 / x86-64 / x64) is an extension of [x86](#) architecture with full backward compatibility. There are many variants of its name and it causes some confusion, but all these names mean the same thing: x86-64, AA-64, Hammer Architecture, AMD64, Yamhill Technology, EM64T, IA-32e, Intel 64, x64. To learn how so many names appeared see the article in Wikipedia: "[X86-64](#)".

You should understand that IA-64 and Intel 64 are absolutely different, incompatible with each other, microprocessor architectures. Within the scope of this course we will consider only Intel 64 (x64 / AMD64) architecture as the most popular among applied Windows software developers. Accordingly, when we mention Windows operating system, we will mean its 64-bit versions for Intel 64 architecture. For example: Windows XP Professional x64 Edition, Windows Vista x64, Windows 7 x64. The program model Intel 64 available to a programmer in a 64-bit Windows is called [Win64](#), for short.

### Intel 64 architecture

The information given here is based on the first volume of the documentation "[AMD64 Architecture Programmer's Manual, Volume 1. Application Programming](#)".

The architecture [Intel 64](#) we are considering here, is a simple yet powerful extension of the obsolete commercial architecture x86 with backward compatibility. It adds the 64-bit address space and extends resources to support higher performance of recompiled 64-bit programs. The architecture supports obsolete 16-bit and 32-bit code of applications and operating systems without modifying or recompiling them.

The need of a 64-bit architecture is determined by the applications that require a larger address space. First of all, these are high-performance servers, data managers, CAD and, of course, games. These applications will get great benefits from the 64-bit address space and larger number of registers. Few registers available in the obsolete x86 architecture limit performance of computing tasks. The increased number of registers provides the necessary performance for many applications.

Let us point out the main advantages of the architecture x86-64:

- the 64-bit address space;
- an extended register set;

- a command set familiar to developers;
- the capability to launch obsolete 32-bit applications in a 64-bit operating system;
- the capability to use 32-bit operating systems.

## 64-bit operating systems

Nearly all modern operating systems have versions for Intel 64 architecture. For example, Microsoft ships Windows XP x64. Large UNIX developers also ship 64-bit versions, for example, Linux Debian 3.5 x86-64. But it does not mean that the whole code of such a system is 64-bit. Some parts of the operating system and many applications may well remain 32-bit because Intel 64 provides backward compatibility. Thus, the 64-bit version of Windows uses a special mode [WoW64](#) (Windows-on-Windows 64) that translates the calls of 32-bit applications to the resources of the 64-bit operating system.

## Address space

Although a 64-bit processor can theoretically address 16 Ebytes of memory ( $2^{64}$ ), Win64 now supports only 16 Tbytes ( $2^{44}$ ). There are some reasons for that. Contemporary processors can provide access only to one Tbyte ( $2^{40}$ ) of physical memory. The architecture (but not the hardware part) can extend this space up to 4 Pbytes ( $2^{52}$ ). But in this case you need an immense amount of memory for the page tables representing it.

Besides the limitations described above, the size of memory available in every particular version of the 64-bit Windows depends upon the commercial reasons of Microsoft. Different Windows versions have different limitations which are illustrated in the table.

64-bit versions of Windows	memory
Windows XP Professional	128 Gbyte
Windows Server 2003, Standard	32 Gbyte
Windows Server 2003, Enterprise	1 Tbyte
Windows Server 2003, Datacenter	1 Tbyte
Windows Server 2008, Datacenter	2 Tbyte
Windows Server 2008, Enterprise	2 Tbyte
Windows Server 2008, Standard	32 Gbyte
Windows Server 2008, Web Server	32 Gbyte
Vista Home Basic	8 Gbyte
Vista Home Premium	16 Gbyte
Vista Business	128 Gbyte
Vista Enterprise	128 Gbyte
Vista Ultimate	128 Gbyte

*Table 1 - The amounts of memory supported in different Windows versions*

## Win64 program model

Like in [Win32](#), the size of a page in Win64 is 4 Kbytes. The first 64 Kbytes of the address space are never displayed, so the lowest correct address is 0x10000. Unlike Win32, system DLL's take more than 4 Gbytes.

Compilers for Intel 64 have one peculiarity: they can use registers with great efficiency to pass parameters into functions instead of using the stack. This allowed the Win64 architecture developers to get rid of such a notion as a [calling convention](#). In Win32, you may use various conventions: `__stdcall`, `__cdecl`, `__fastcall`, etc. In Win64, there is only one calling convention. Here is an example of how four arguments of *integer* type are passed through registers:

- RCX: the first argument
- RDX: the second argument
- R8: the third argument
- R9: the fourth argument

The arguments following the first four integer ones are passed through the stack. To pass float arguments XMM0-XMM3 registers are used as well as the stack.

The difference in the calling conventions makes it impossible to use both 64-bit and 32-bit code in one program. In other words, if an application has been compiled for the 64-bit mode, all the libraries (DLL) being used must also be 64-bit.

Passing parameters through registers is one of the innovations that make 64-bit programs faster than 32-bit ones. You may achieve an additional performance gain using 64-bit data types. We will tell you about it in the next lesson.

*The course authors: Andrey Karpov ([karpov@viva64.com](mailto:karpov@viva64.com)), Evgeniy Ryzhkov ([evg@viva64.com](mailto:evg@viva64.com)).*

*The rightholder of the course "Lessons on development of 64-bit C/C++ applications" is OOO "Program Verification Systems". The company develops software in the sphere of source program code analysis. The company's site: <http://www.viva64.com>.*

*Contacts: e-mail: [support@viva64.com](mailto:support@viva64.com), Tula, 300027, PO box 1800.*

## Lesson 2. Support of 32-bit applications in the 64-bit Windows environment

Before we start discussing the topic of developing 64-bit program code, let us speak about backward compatibility of 64-bit Windows versions with 32-bit applications. Backward compatibility is arranged through the mechanisms implemented in [WoW64](#).

**WoW64** (Windows-on-Windows 64-bit) is a subsystem of Windows operating system that allows you to execute 32-bit applications on all the 64-bit versions of Windows.

The WoW64 subsystem does not support the following programs:

- programs compiled for 16-bit operating systems;
- kernel-mode programs compiled for 32-bit operating systems.

## Indirect expenses

Different processor architectures have a bit different WoW64. For example, the 64-bit Windows version developed for Intel Itanium 2 processor employs WoW64 to emulate [x86](#) instructions. This emulation is rather resource-intensive in comparison to WoW64 for [Intel 64](#) architecture because the system has to switch from the 64-bit mode to compatibility mode when executing 32-bit programs.

WoW64 on Intel 64 (AMD64 / x64) does not require instruction emulation. In this case the WoW64 subsystem emulates only the 32-bit environment through an additional layer between a 32-bit application and the 64-bit Windows API. In some places this layer is thin, in others a bit thicker. For an average program, you may expect 2% performance penalty because of this layer. For some programs, it can be larger. Two per cent is not very much but keep in mind that 32-bit applications work a bit slower under the 64-bit Windows than in the 32-bit environment.

Compilation of 64-bit code does not only allow you to avoid using WoW64 but also gives you an additional performance gain. This is explained by architectural modifications in the microprocessor such as an increased number of general-purpose registers. For an average program, you may expect a 5-15% performance gain after mere recompilation.

## Benefits of the 64-bit environment for 32-bit programs

Because of the WoW64 layer, 32-bit programs are less efficient in the 64-bit environment than in their native 32-bit one. But still simple 32-bit applications can get one benefit of being executed in the 64-bit environment. Maybe you know that a program built with the switch `"/LARGEADDRESSAWARE:YES"` can allocate up to 3 Gbytes of memory if a 32-bit Windows is launched with the switch `"/3gb"`. Well, the same 32-bit program built on a 64-bit system can allocate almost 4 Gbytes of memory (in practice it is usually about 3.5 Gbytes).

## Redirections

The WoW64 subsystem isolates 32-bit programs from 64-bit ones by redirecting calls to files and the register. It helps to keep 32-bit programs from accidentally accessing the data of 64-bit ones. For example, a 32-bit application that launches a DLL file from the catalogue `"%systemroot%\System32"` can accidentally address a 64-bit DLL which is incompatible with the 32-bit program. To avoid this, the WoW64 subsystem redirects the access from the folder `"%systemroot%\System32"` into the folder `"%systemroot%\SysWOW64"`. This redirection helps you avoid compatibility errors because the 32-bit application will need a special DLL file created to work with 32-bit applications.

To learn more about the mechanisms of file system and register redirection see MSDN section "[Running 32-bit Applications](#)".

## Why cannot 32-bit DLL's be used in a 64-bit program? Is there a way to evade this limitation?

It is impossible to load a 32-bit DLL from a 64-bit process and execute its code. It is impossible due to the design of 64-bit systems. It is impossible fundamentally. And no tricks and undocumented means will help you. To do this you will have to load and initialize WoW64, not to speak of the kernel structures. Actually, it means that a 64-bit process must be made 32-bit "on the fly". This topic is described more thoroughly in the post "[Why can't you thunk between 32-bit and 64-bit Windows?](#)". The only thing we can recommend is to create a surrogate process and work with it through the COM technology. You may read about it in the article "[Accessing 32-bit DLLs from 64-bit code](#)".

But it is quite easy to load resources from a 32-bit DLL into a 64-bit process. You may do it specifying the flag `LOAD_LIBRARY_AS_DATAFILE` when calling `LoadLibraryEx`.

## Gradual renunciation of 32-bit software support

It will be quite natural if Microsoft company will stimulate the move to 64-bit systems by gradually canceling the support of 32-bit programs in some versions of Windows operating system. Of course it will be a very slow process but the first steps in this direction have been already made.

Many administrators know about a relatively new installation and operation mode of the server version of the operating system called Server Core. It is that very mode the participants of "Windows vs Linux" wars have been speaking of for a long time. One of the reasons that adherents of using Linux on servers referred to was the capability to install the server operating system without graphical interface (GUI). But here is such a capability in Windows Server too. Now, if you install the system in this mode, you will get only the command line without user interface.

This capability (Server Core installation) appeared in Windows Server 2008. Yet in Windows Server 2008 R2 there is another innovation that brings closer the 64-bit future. Support of 32-bit applications [is now optional](#) and you may enable or disable it when installing Windows Server 2008 R2 (Server Core). Moreover, this option is disabled by default. So when trying to launch a 32-bit application in Server Core mode, you will get a message telling you that it is impossible. Of course you may add 32-bit application support:

```
start /w ocsetup ServerCore-WOW64
```

In the usual (Full Installation) mode, execution of 32-bit applications is enabled by default, but not in Server Core.

The tendency is obvious. It will be more and more rational to create 64-bit versions of applications in time as they will be able to work on more operating system versions.

## Additional information

Alexey Pahunov's Russian blog is also a very interesting source of information on WoW64: <http://blog.not-a-kernel-guy.com/>. Alexey is Microsoft company's worker and he personally participates in developing the WoW64 subsystem.

*The course authors: Andrey Karpov ([karpov@viva64.com](mailto:karpov@viva64.com)), Evgeniy Ryzhkov ([evg@viva64.com](mailto:evg@viva64.com)).*

*The rightholder of the course "Lessons on development of 64-bit C/C++ applications" is OOO "Program Verification Systems". The company develops software in the sphere of source program code analysis. The company's site: <http://www.viva64.com>.*



Contacts: e-mail: [support@viva64.com](mailto:support@viva64.com), Tula, 300027, PO box 1800.

## Lesson 3. Porting code to 64-bit systems. The pros and cons

You should begin studying [64-bit](#) systems with the question "How much rational it will be to recompile a project for a 64-bit system?". You must answer this question but take your time and think it over. On the one hand, you might lag behind your rivals failing to offer 64-bit solutions on market. On the other hand, you might waste your time developing a 64-bit application that will have no competitive advantages.

Here are some factors that will help you make a choice.

### Application life-cycle

You should not create a 64-bit version of an application with a short life-cycle. The [WoW64](#) subsystem allows obsolete 32-bit applications to work rather well on 64-bit Windows systems. It is unreasonable to make a program 64-bit if you stop maintaining it in 2 years. The practice shows that the move to 64-bit Windows versions will be very slow and smooth. Perhaps most of your users will use only the 32-bit version of your program solution in the nearest future. You should keep in mind that this course was written in 2009 when most users were working with 32-bit versions of operating systems. But in time 32-bit programs will look more and more unnatural and outdated.

If you plan a prolonged development and maintenance of your program product, you should start working on its 64-bit version. Of course you should take your time but keep in mind that the later you have a full 64-bit version, the more problems you are to encounter while maintaining such an application installed on 64-bit Windows versions.

### Application performance requirements

After being recompiled for a 64-bit system a program can use huge amounts of memory and its speed will increase in 5-15%. 5-10% of speed gain is achieved due to architectural features of the 64-bit processor, for example, a larger number of registers. And another 1-5% performance gain is determined by the absence of the WoW64 layer that translates calls between 32-bit applications and the 64-bit operating system.

For example, Adobe company says that a new 64-bit "Photoshop CS4" is 12% faster than its 32-bit version".

Applications involving large memory amounts can expect a great performance gain. These are graphical editors, CAD-systems, GSI CAD, databases and packages for modeling various processes. The capability to store all the data in memory and therefore avoid additionally loading them from the hard disk may increase the speed of such applications not in some per cent but in several times.

For example, take Alfa-Bank that integrated an Itanium 2 based platform into their IT-infrastructure. The growth of their investment business had caused the system to fail to manage the increasing load on the current configuration any more: the number of customer support delays sometimes got very critical. The analysis of the situation showed that the bottleneck of the system had nothing to do with processors' performance but it was the limitation of the 32-bit architecture regarding the memory subsystem that

allowed using not more than 4 Gbytes of the server address space. The database size was more than 9 Gbytes. It had been used very intensively and that caused a critical loading of the input-output subsystem. Alfa-Bank decided to buy a cluster of two four-processor servers based on Itanium 2 with 12 Gbytes of memory. This decision allowed them to get the necessary performance and fault-tolerance level. As the company representatives say, introduction of Itanium 2 based servers allowed them to eliminate serious issues and manage to save much money.

## **Using third-party libraries in a project**

Before planning the work on developing the 64-bit version of your product, make it out if there are 64-bit versions of libraries and components it employs. You should also find out the pricing policy regarding the 64-bit versions of the libraries. All this you may learn on the site of library developers. If there is no support for the libraries, search for alternative means supporting 64-bit systems beforehand.

## **Dependence of third-party developers upon your libraries**

If you are developing libraries, components or other items intended for third-party developers to create software with, you must be quick in creating the 64-bit version of your product. Otherwise, your customers interested in 64-bit versions will have to search for other solutions. For example, some software and hardware security developers appeared to be very late in creating 64-bit programs and it made some of their clients choose other tools to protect their software products.

There is one more benefit of releasing a 64-bit version of a library: you may sell it as a separate product. Thus, your customers who wish to create both 32-bit and 64-bit applications will have to buy 2 different licenses. For example, Spatial Corporation company sticks to such a policy when selling their library Spatial ACIS.

## **16-bit applications**

If your solutions still have 16-bit modules, you must get rid of them. 64-bit Windows versions do not support 16-bit applications.

I should explain one thing here related to using 16-bit installers. They are still used to install some 32-bit applications. There exists a special mechanism that replaces some of the most popular 16-bit installers with their more contemporary versions on the fly. It might make you think that 16-bit programs still work in the 64-bit environment, but it is a mistake, please, keep it in mind.

## **Assembler code**

Do not forget that presence of large assembler code fragments make it much more expensive to create the 64-bit version of an application.

## **Toolkit**

If you have decided to create the 64-bit version of your product relying on the factors mentioned above and are ready to spend time on it, the success is not guaranteed yet. You should have all the necessary tools for that and here you might encounter some very unpleasant things.



The most obvious yet most serious problem is absence of a 64-bit compiler. When we were writing this text (2009) there was no 64-bit C++ Builder compiler by Embarcadero yet. Its release was expected by the end of 2009. You cannot evade this problem unless you rewrite the whole project employing, for example, Microsoft Visual Studio. But while everything is clear in case of compiler absence, other similar issues might be not so obvious and occur only at the step of porting the project to a new architecture. You should make a research beforehand to find out if you can get all the necessary components to implement the 64-bit version of your product. You might face unpleasant surprises.

While making a decision, please keep in mind the last very important factor we have not mentioned here: the price of modifying your program code to compile it in the 64-bit mode. We will tell you how to estimate this price in one of the following lessons. It may be very high and must be considered in planning and scheduling.

*The course authors: Andrey Karpov ([karpov@viva64.com](mailto:karpov@viva64.com)), Evgeniy Ryzhkov ([evg@viva64.com](mailto:evg@viva64.com)).*

*The rightholder of the course "Lessons on development of 64-bit C/C++ applications" is OOO "Program Verification Systems". The company develops software in the sphere of source program code analysis. The company's site: <http://www.viva64.com>.*

*Contacts: e-mail: [support@viva64.com](mailto:support@viva64.com), Tula, 300027, PO box 1800.*

## Lesson 4. Creating the 64-bit configuration

### Compiler

At first you should make sure that the Visual Studio edition you are using allows building 64-bit code. If you want to develop 64-bit applications using the latest (at the moment of writing this course) Visual Studio 2008 version, here is a table that will help you understand what Visual Studio edition you need.

	Microsoft Visual C++ Express Edition	Visual Studio 2008 Standard	Visual Studio 2008 Professional	Visual Studio 2008 Team System
32-bit x86 compiler	YES	YES	YES	YES
64-bit x64 compiler and cross-compiler		YES	YES	YES
64-bit Itanium compiler and cross-compiler				YES

*Table 1 – Capabilities of different Visual Studio 2008 editions*

If the Visual Studio edition you are using allows creating 64-bit code, you should check if the 64-bit compiler is installed. Figure 1 shows the page of installing Visual Studio 2008 components where installation of the 64-bit compiler is disabled.

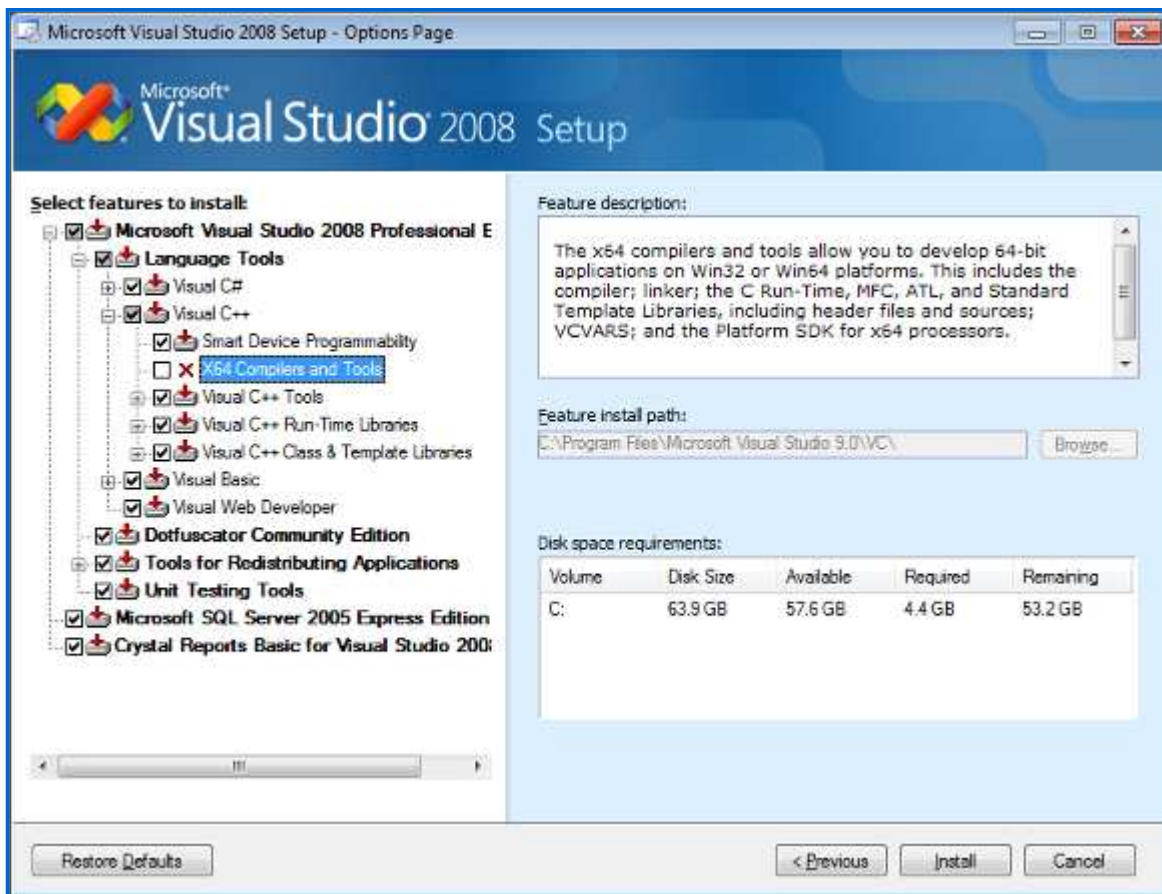


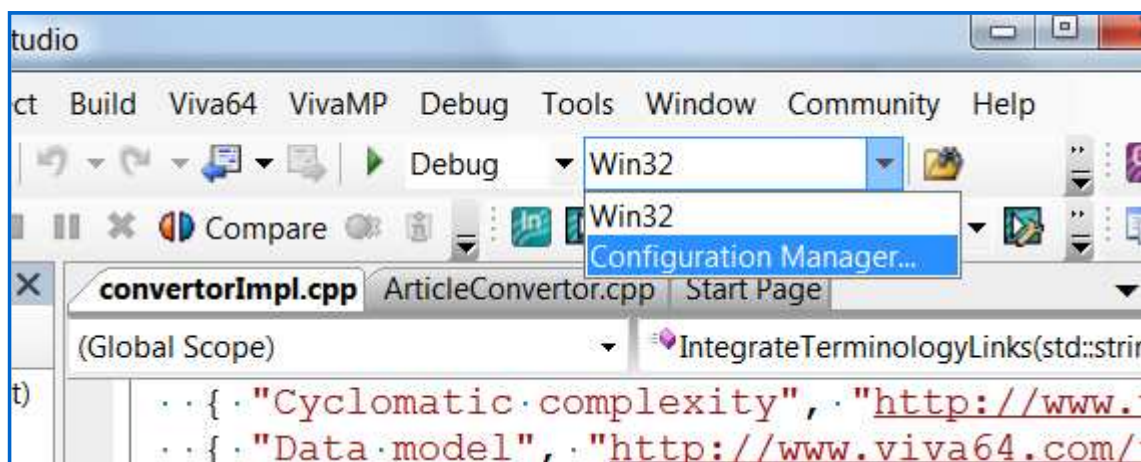
Figure 1 – The 64-bit compiler is disabled when installing Visual Studio 2008

## Creating the 64-bit configuration

Creating the 64-bit version of a project in Visual Studio 2005/2008 is a rather simple procedure. Difficulties will appear later at the step of building the new configuration and searching for errors in it. To create a 64-bit configuration you should make the following 4 steps:

### Step 1

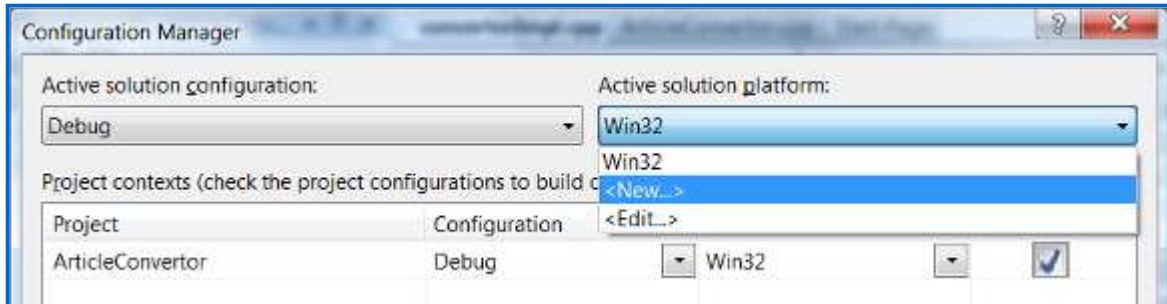
Open the configuration manager as shown in Figure 2:



*Figure 2 – Launching the configuration manager*

## Step 2

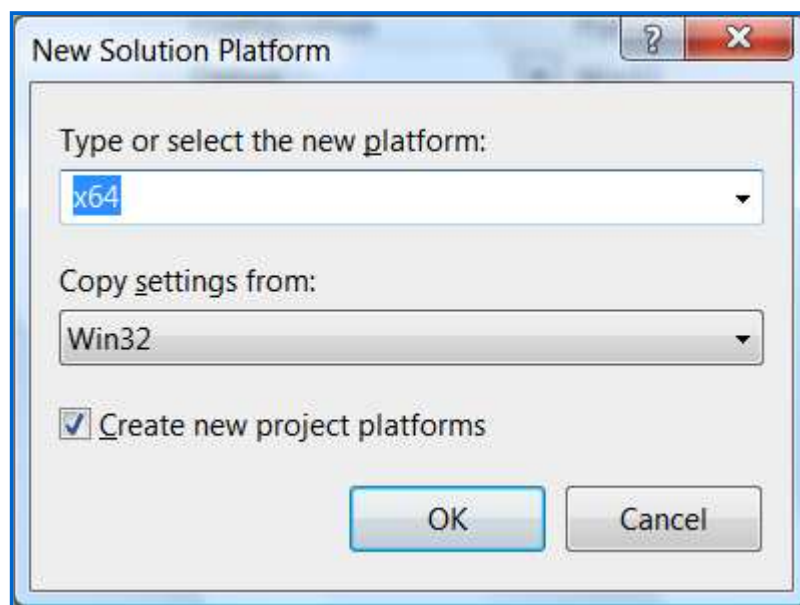
Choose support of the new platform in the configuration manager (Figure 3):



*Figure 3 – Creating a new configuration*

## Step 3

Choose the 64-bit platform (x64) and take the 32-bit version settings as the base (Figure 4). Visual Studio environment will automatically modify those settings that impact the build mode.



*Figure 4 – Choosing x64 as the platform and loading the Win32 configuration as the base*

## Step 4

You have added the new configuration and now may select the 64-bit configuration version and start compiling the 64-bit application. Figure 5 shows how to choose the 64-bit building configuration.

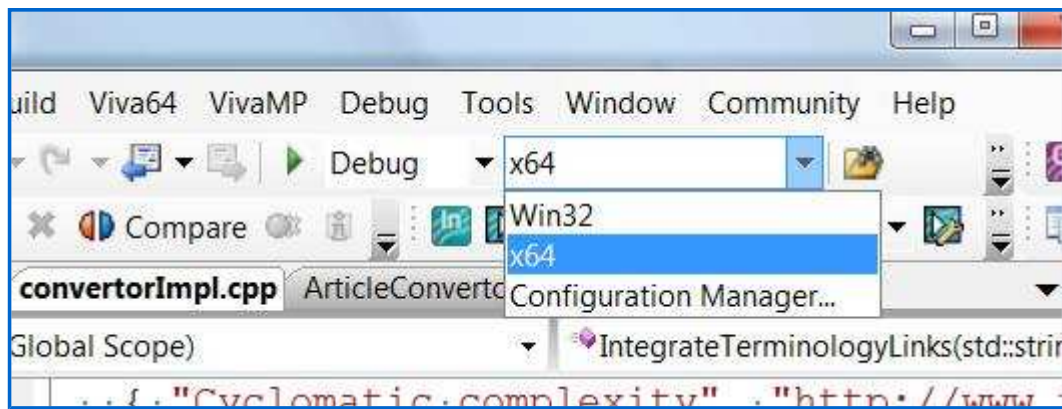


Figure 5 – Now you have both the 32-bit and 64-bit configurations

## Modifying parameters

If you are lucky, you will not have to adjust the 64-bit project. But this depends upon the project, its complexity and the number of libraries being used. The only thing you should modify right away is the stack size. If your project uses the stack of the default size, i.e. 1 Mbyte, you should change it to 2 Mbytes for the 64-bit version. It is not necessary but it is better to secure yourself from possible issues beforehand. If you use the stack of a size different from that by default, you should make it twice larger for the 64-bit version. To do it find and change the parameters *Stack Reserve Size* and *Stack Commit Size* in the project settings (see Figure 6).

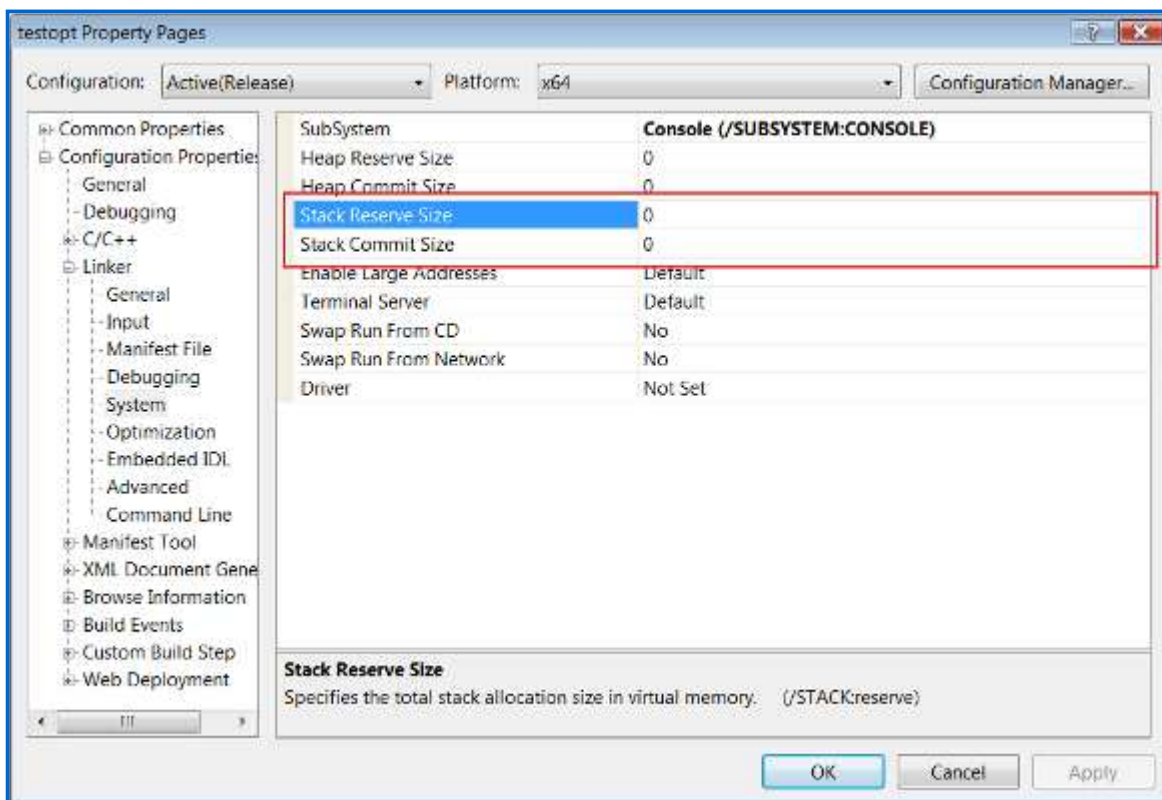


Figure 6 – Location of project settings defining the stack size

## What next?

Having the 64-bit configuration of a project does not mean that it will compile well and work at all. The process of compilation and detection of hidden errors will be discussed in the next lessons.

*The course authors: Andrey Karpov ([karpov@viva64.com](mailto:karpov@viva64.com)), Evgeniy Ryzhkov ([evg@viva64.com](mailto:evg@viva64.com)).*

*The rightholder of the course "Lessons on development of 64-bit C/C++ applications" is OOO "Program Verification Systems". The company develops software in the sphere of source program code analysis. The company's site: <http://www.viva64.com>.*

*Contacts: e-mail: [support@viva64.com](mailto:support@viva64.com), Tula, 300027, PO box 1800.*

## Lesson 5. Building a 64-bit application

We would like to warn the readers right away that it is impossible to describe the process of building a 64-bit application in every detail. Any project has its own unique settings, so you must be very attentive when adapting them for a 64-bit system. The lesson discusses only the common steps important for any project. These steps will tell you where to begin.

### Libraries

Before trying to build your 64-bit application, make sure that all the necessary versions of 64-bit libraries are installed and paths to them are correct. For example, 32-bit and 64-bit library files with "lib" extension usually differ and are situated in different catalogues. Fix the bugs if any.

*Note. If libraries are presented in the form of the source code, there must be the 64-bit configuration of the project. Keep in mind that you are risking to infringe license agreements when modifying a library to build its 64-bit version by yourself.*

### Assembler

Visual C++ does not support the 64-bit inline assembler. You must either use an external 64-bit assembler (for example, MASM) or rewrite the assembler code in C/C++.

### Examples of compilation errors and warnings

On starting to build the project you will encounter many compilation errors and warnings related to explicit and implicit type conversions. We would like to show you an example of such an error. Here is a code:

```
void foo(unsigned char) {}
void foo(unsigned int) {}

void a(const char *str)
{
    foo(strlen(str));
}
```

This code successfully compiles in the 32-bit mode, but in the 64-bit mode, Visual C++ compiler will generate the warning:



```
error C2668: 'foo' : ambiguous call to overloaded function
.\xxxx.cpp(16): could be 'void foo(unsigned int)'
.\xxxx.cpp(15): or 'void foo(unsigned char)'
while trying to match the argument list '(size_t)'
```

The function `strlen()` returns the type [size\\_t](#). On a 32-bit system, the type `size_t` coincides with the type `unsigned int` and the compiler chooses the function "void foo(unsigned int)" to call. In the 64-bit mode, the types `size_t` and `unsigned int` do not coincide. The type `size_t` becomes 64-bit while the type `unsigned int` remains 32-bit. As a result, the compiler does not know which of the `foo()` functions to prefer.

Now consider an example of a warning generated by Visual C++ compiler when building code in the 64-bit mode:

```
CArray<char, char> v;
int len = v.GetSize();

warning C4244: 'initializing' : conversion from 'INT_PTR' to 'int',
possible loss of data
```

The function `GetSize()` returns the type [INT\\_PTR](#) that coincides with the type `int` in a 32-bit code. In a 64-bit code, the type `INT_PTR` is 64-bit and it is implicitly converted to the 32-bit `int` type. The values of more significant bits get lost during this process and the compiler warns you about it. An implicit type conversion may cause an error if the number of the array items exceeds `INT_MAX`. To eliminate the warning and the possible error you should assign the type `INT_PTR` or [ptrdiff\\_t](#) to "len" variable.

**Do not correct warnings** until you have learned the 64-bit error patterns. You might accidentally hide an error failing to correct it and make it more difficult to detect further. You will learn about the patterns of 64-bit errors and methods of detecting and correcting them in the next lessons. You may also see the following articles: "[20 issues of porting C++ code on the 64-bit platform](#)", "[A 64-bit horse that can count](#)".

## size\_t and ptrdiff\_t types

As most compilation errors and warnings are related to data type incompatibility, we should consider two types - `size_t` and `ptrdiff_t` - which are most relevant to us regarding the process of 64-bit code creation. If you are using Visual C++ compiler, these types are integrated into it and you will not need the library files. But if you are using GCC, you will need the header file "stddef.h".

**size\_t** is a C/C++ base unsigned integer type. It is the type of the result returned by `sizeof` operator. The size of the type is chosen so that it could store the maximum size of a theoretically possible array of any type. For example, `size_t` is 32-bit on a 32-bit system and 64-bit on a 64-bit one. In other words, you may safely store a pointer in a variable of `size_t` type. Pointers to class functions are an exception but this is a different topic. The type `size_t` is usually used in loop counters, to index arrays, to store sizes and in address arithmetic. The following types are analogous to `size_t`: `SIZE_T`, `DWORD_PTR`, `WPARAM`, `ULONG_PTR`. Although you may store a pointer in `size_t`, it is better to use another unsigned integer type `uintptr_t` for that - its name reflects its capability. The types `size_t` and `uintptr_t` are synonyms.

**ptrdiff\_t** is a C/C++ base signed integer type. Its size is chosen so that it could store the maximum size of a theoretically possible array of any type. This type will be 32-bit on a 32-bit system and 64-bit on a 64-bit one. Like `size_t`, a variable of `ptrdiff_t` type can safely store a pointer except for a pointer to a class function. The type `ptrdiff_t` is also the result of an expression where one pointer is subtracted from



another "ptr1-ptr2". The type `ptrdiff_t` is usually used in loop counters, to index arrays, to store sizes and in address arithmetic. Its analogues are: `SSIZE_T`, `LPARAM`, `INT_PTR`, `LONG_PTR`. The type `ptrdiff_t` has a synonym `intptr_t` whose name reflects it more clearly that it can store a pointer.

The sizes `size_t` and `ptrdiff_t` were created to perform correct [address arithmetic](#). It has been considered for a long time that the size of `int` coincides with the size of the machine word (processor capacity) and it can be used as indexes and to store sizes of objects and pointers. So, address arithmetic was also built with `int` and unsigned types. The type `int` is used in most education materials on C and C++ programming in loop bodies and as indexes. The following example is almost a canon:

```
for (int i = 0; i < n; i++)
    a[i] = 0;
```

As processors were developing and their capacity increasing, it became unreasonable to further increase the capacities of `int` type. There are a lot of reasons for that: the purposes of saving memory being used, maximum compatibility, etc. As a result, several [data models](#) appeared describing the relations of the base C and C++ types. So it is not so easy now to choose a type for a variable to store a pointer or object size. `size_t` and `ptrdiff_t` types appeared to become the smartest solution of this problem. They can certainly be used in address arithmetic. Now, the following code must become a canon:

```
for (ptrdiff_t i = 0; i < n; i++)
    a[i] = 0;
```

It is this code that can provide safety, good portability and performance. You will learn from the next lessons why.

The types `size_t` and `ptrdiff_t` we have described may be called [memsize](#)-types. The term "memsize" appeared as an attempt to briefly name all the types that can store sizes of pointers or indexes of the largest arrays. By memsize-types you should understand all the simple C/C++ data types that are 32-bit on a 32-bit architecture and 64-bit on a 64-bit one. Here are examples of memsize-types: `size_t`, `ptrdiff_t`, pointers, `SIZE_T`, `LPARAM`.

*The course authors: Andrey Karpov ([karpov@viva64.com](mailto:karpov@viva64.com)), Evgeniy Ryzhkov ([evg@viva64.com](mailto:evg@viva64.com)).*

*The rightholder of the course "Lessons on development of 64-bit C/C++ applications" is OOO "Program Verification Systems". The company develops software in the sphere of source program code analysis. The company's site: <http://www.viva64.com>.*

*Contacts: e-mail: [support@viva64.com](mailto:support@viva64.com), Tula, 300027, PO box 1800.*

## Lesson 6. Errors in 64-bit code

Even if you correct all compilation errors and warnings, it does not mean that a 64-bit application will work well. So it is the description and diagnosis of 64-bit errors that we will deal with in the most lessons of our course. And one more thing - do not rely on the switch [/Wp64](#) which is described by many people (often unreasonably) in forum discussions as a wonderful tool able to find 64-bit errors.

### /Wp64 switch

The switch */Wp64* allows programmers to find some issues that may occur when compiling code for 64-bit systems. The check is implemented in this way: the types marked with the key word `__w64` in 32-bit code are interpreted as 64-bit types while being checked.

For example, here is a code:

```
typedef int MyInt32;

#ifdef _WIN64
    typedef __int64 MySSizet;
#else
    typedef int MySSizet;
#endif

void foo() {
    MyInt32 value32 = 10;
    MySSizet size = 20;
    value32 = size;
}
```

The expression "`value32 = size;`" will lead to value cutting on a 64-bit system and therefore to a possible error. We want to diagnose this issue. But when we try to compile the 32-bit application, everything is correct and there is no warning.

To get ready to move the application to 64-bit systems we need to add the switch */Wp64* and the key word `__w64` when defining the type `MySSizet` in the 32-bit version. After that the code looks so:

```
typedef int MyInt32;

#ifdef _WIN64
    typedef __int64 MySSizet;
#else
    typedef int __w64 MySSizet; // Add __w64 keyword
#endif

void foo() {
    MyInt32 value32 = 10;
    MySSizet size = 20;
    value32 = size; // C4244 64-bit int assigned to 32-bit int
}
```

Now we get the warning *C4244* that will help us in porting the code to a 64-bit platform.

Note that the switch */Wp64* is ignored in the 64-bit compilation mode because all the types already have the necessary size and the compiler performs the necessary checking. So, as you can see, we will get the warning *C4244* when compiling the 64-bit version even if the switch */Wp64* is disabled.

So, the switch */Wp64* helped developers get somehow ready to use the 64-bit compiler while working with 32-bit applications. All warnings revealed with the help of */Wp64* will turn into compilation errors or remain warnings when building the 64-bit code. And that is all aid you may expect from the switch */Wp64* in detecting errors.

By the way, the switch */Wp64* is considered deprecated in Visual Studio 2008 because it is high time we started to compile 64-bit applications instead of going on to get ready for it.

## 64-bit errors

When we speak of 64-bit errors, we mean those cases when a code fragment that works well in the 32-bit version of an application causes errors after being recompiled in the 64-bit mode. 64-bit errors occur most frequently in the following kinds of code fragments:

- code based on wrong assumptions about type sizes (for example, an assumption that the pointer size is always 4 bytes);
- code processing large arrays whose size is more than 2 Gbytes on 64-bit systems;
- code responsible for data writing and reading;
- code containing bit operations;
- code with complex address arithmetic;
- obsolete code;
- and so on.

In fact, all errors occurring in the code when it is recompiled for 64-bit systems arise from inaccurate compliance with C/C++ standard ideology. But we do not find it very reasonable to follow this recommendation: "write correct programs and there will be no 64-bit errors". One cannot argue against it but it has little relevance to real projects. There is much C/C++ code in the world that has been written for many decades. The purpose of our lessons is to arrange all the 64-bit errors into a set of patterns that will help you detect defects and instruct you how to eliminate them.

## Examples of 64-bit errors

We will speak a lot about 64-bit errors in future but here are two examples for you to understand what these errors are.

The first is an example of using the magic constant 4 that serves as the size of a pointer what is incorrect in 64-bit code. Note that this code worked quite well in the 32-bit version and was not diagnosed as dangerous by the compiler.

```
size_t pointersCount = 100;
int **arrayOfPointers = (int **)malloc(pointersCount * 4);
```

The second is an example of an error in the data reading mechanism. This code is correct in the 32-bit version and the compiler does not react to it. But this code fails to correctly read the data saved by the 32-bit version of the program.

```
size_t PixelCount;
fread(&PixelCount, sizeof(PixelCount), 1, inFile);
```

## A comment for sophisticated programmers

I would like to comment right away upon the 64-bit error patterns and error examples that will be discussed in many following lessons. People often argue that actually these are not errors related to 64 bits but the errors arising from an incorrectly written and badly portable code. And they also say that many errors can be found when porting code not only to the 64-bit architecture but simply to any architecture where the base types have other sizes.

Yes, that is right! We keep this in mind. But our goal is not to study the issue of code portability as such. In these lessons we are going to solve a particular local task - to help developers in mastering 64-bit platforms that become more and more popular.

When speaking of 64-bit error patterns we will consider examples of code that is correct on 32-bit systems but may cause faults when being ported to the 64-bit architecture.

*The course authors: Andrey Karpov ([karpov@viva64.com](mailto:karpov@viva64.com)), Evgeniy Ryzhkov ([evg@viva64.com](mailto:evg@viva64.com)).*

*The rightholder of the course "Lessons on development of 64-bit C/C++ applications" is OOO "Program Verification Systems". The company develops software in the sphere of source program code analysis. The company's site: <http://www.viva64.com>.*

*Contacts: e-mail: [support@viva64.com](mailto:support@viva64.com), Tula, 300027, PO box 1800.*

## Lesson 7. The issues of detecting 64-bit errors

There are various techniques of detecting errors in program code. Let us consider the most popular ones and see how efficient they are in finding 64-bit errors.

### Code review

The oldest and the most proved and reliable approach to error search is code review. This method relies on reading the code by several developers together following some rules and recommendations described in the book by Steve McConnell "Code Complete" ([Steve McConnell, "Code Complete"](#)). Unfortunately, this method cannot be applied to large-scale testing of contemporary program systems due to their huge sizes.

Code review may be considered in this case rather a good means of education and avoiding 64-bit errors in a new code being developed. But this method will be too expensive and therefore unacceptable in searching for the already existing errors. You would have to view the code of the whole project to find all 64-bit errors.

### Static code analysis

The means of [static code analysis](#) will help those developers who appreciate the regular code review but do not have enough time to do that. The main purpose of static code analysis is to reduce the amount of code needed to be viewed by a human and therefore reduce the time of code review. Rather many programs refer to static code analyzers which have implementations for various programming languages and provide a lot of various functions from simple code alignment control to complex analysis of potentially dangerous fragments. The advantage of static analysis is its good scalability. You can test a project of any size in reasonable time with its help. And testing the code with static analyzer regularly will help you detect many errors at the stage of only writing the code.

The static analysis technique is the most appropriate method to detect 64-bit errors. Further, when discussing 64-bit error patterns, we will show you how to diagnose these errors using Viva64 analyzer included into [PVS-Studio](#). In the next lesson you will learn in more detail about the static analysis methodology and PVS-Studio tool.

## White box method

By the white box method we will understand the method of executing the maximum available number of different code branches using a debugger or other tools. The more code is covered during the analysis, the more complete the testing is. Also, the white box testing is sometimes understood as simple debugging of an application in order to find some known error. It became impossible a long time ago to completely test the whole program code with the white box method due to huge sizes of contemporary applications. Nowadays, the white box method is convenient to use when an error is found and you want to find out what has caused it. Some programmers oppose the white box technique denying the efficiency of real-time program debugging. The main reason they refer to is that enabling a programmer to watch the process of program execution and change it along the way leads to an unacceptable programming approach implying correction of code by the trial-and-error method. We are not going to discuss these debates but I would like to note that the white box testing is too expensive to use for enhancing the quality of large program systems anyway.

It must be evident to you that complete debugging of an application for the purpose of detecting 64-bit errors is unreal just like the complete code review.

We should also note that the step-by-step debugging might be impossible when debugging 64-bit applications that process large data arrays. Debugging of such applications may take much more time. So you should consider using logging systems or some other means to debug applications.

## Black box method (unit-test)

The black box method has shown much better results. Unit tests refer to this type of testing. The working principle of this technique is writing a set of tests for separate units and functions that checks all the main modes of their operation. Some authors refer unit-testing to the white box method because it relies on knowledge of the program organization. But we think that functions and units being tested should be considered black boxes because unit tests do not take into account the inner organization of a function. This viewpoint is supported by an approach when tests are developed before the functions themselves are written and it provides an increased level of the control over their functionality in terms of specification.

Unit tests have proved to be efficient in developing both simple and complex projects. One of the advantages of unit testing is that you may check if all the changes introduced into the program are correct right along the development process. They try to make it so that tests are run in only a few minutes - it allows the developer who has modified the code to see an error and correct it right away. If it is impossible to run all the tests at once, long-term tests are usually launched separately, for example, at night. It also contributes to a quick detection of errors, at least in the next morning.

When using unit tests to search for 64-bit errors, you are likely to encounter some unpleasant things. Seeking to make quick tests, programmers try to involve a small amount of calculations and data to be processed while developing them. For example, when you develop a test for the function searching for an array item, it does not matter if there will be 100 or 10 000 000 items. A hundred of items is enough but when the function processes 10 000 000, its speed is greatly reduced. But if you want to develop efficient tests to check this function on a 64-bit system, you will have to process more than 4 billion items! You think that if the function works with 100 items, it will work with billions? No. Here is an example.

```
bool FooFind(char *Array, char Value,
             size_t Size)
{
```

```

    for (unsigned i = 0; i != Size; ++i)
        if (i % 5 == 0 && Array[i] == Value)
            return true;
    return false;
}

#ifdef _WIN64
    const size_t BufSize = 5368709120ui64;
#else
    const size_t BufSize = 5242880;
#endif

int _tmain(int, _TCHAR *) {
    char *Array =
        (char *)calloc(BufSize, sizeof(char));
    if (Array == NULL)
        std::cout << "Error allocate memory" << std::endl;
    if (FooFind(Array, 33, BufSize))
        std::cout << "Find" << std::endl;
    free(Array);
}

```

The error here is in using the type *unsigned* for the loop counter. As a result, the counter is overflowed and an eternal loop occurs when processing a large array on a 64-bit system.

*Note. It might be so that this example will not reveal an error with some settings of the compiler. To understand this strange thing, see the article ["A 64-bit horse that can count"](#).*

As you may see from the example, you cannot rely on obsolete sets of unit tests if your program processes a large data amount on a 64-bit system. You must extend them taking into account possible large data amounts to be processed.

Unfortunately, it is not enough to write new tests. Here we face the problem of the time it will take the modified test set processing large data amounts to complete this work. Consequently, such tests cannot be added to the set you could launch right along the development process. Launching them at night also causes issues. The total time of running all the tests may increase more than ten times. As a result, the test running time may become more than 24 hours. You should keep this in mind and take it very seriously when modifying the tests for the 64-bit version of your program.

## Manual testing

This method can be considered the final step of any development process but you should not take it as a good and safe technique. Manual testing must exist because it is impossible to detect all the errors in the automatic mode or with code review. But you should not fully rely on it either. If a program is low-quality and has a lot of defects, it may take you a long time to test and correct it and still you cannot provide the necessary quality. The only way to get a quality program is to have a quality code. That is why we are not going to consider manual testing as an efficient method of detecting 64-bit errors.

To sum it up, I would like to say that you should not rely on only one of the methods we have discussed. Although static analysis is the most efficient technique of detecting 64-bit errors, a quality application cannot be developed when only a couple of testing methodologies are involved.

*The course authors: Andrey Karpov ([karpov@viva64.com](mailto:karpov@viva64.com)), Evgeniy Ryzhkov ([evg@viva64.com](mailto:evg@viva64.com)).*



*The rightholder of the course "Lessons on development of 64-bit C/C++ applications" is OOO "Program Verification Systems". The company develops software in the sphere of source program code analysis. The company's site: <http://www.viva64.com>.*

*Contacts: e-mail: [support@viva64.com](mailto:support@viva64.com), Tula, 300027, PO box 1800.*

## **Lesson 8. Static analysis for detecting 64-bit errors**

### **Static code analysis**

Static code analysis is a methodology of detecting errors in program code relying on studying the code fragments marked by the static analyzer by the programmer. The marked code fragments are most likely to contain errors of some particular kind.

In other words, a static analysis tool detects those places in the program text which are subject to errors or have bad formatting. Such code fragments are left for the programmer to study them and decide if they must be modified.

Static analyzers may be general-purpose (for example, Microsoft PREFast, Gimpel Software PC-lint, Parasoft C++test) and special-purpose to search for some particular error classes (for example, Chord to verify concurrent Java programs). Usually static analysis tools are rather expensive and require that you know how to use them. They often provide rather flexible yet complicated subsystems of settings and false alarm suppression. Because of this static analyzers are used, as a rule, in companies providing a mature level of software development processes. In exchange for being complicated to use static code analyzers allow programmers to detect a lot of errors at the early stages of program code development. The practice of using static analysis also disciplines programmers and helps managers control young employees.

The main advantage of static code analyzers is an opportunity to greatly reduce the costs of eliminating defects in a program. The earlier an error is detected, the less expensive it is to correct it. Thus, according to the book "Code Complete" by McConnell, correction of an error at the stage of testing the code is five times more expensive than at the stage of designing the code (coding):

	Time Detected				
Time Introduced	Requirements	Architecture	Construction	System Test	Post-Release
Requirements	1	3	5-10	10	10-100
Architecture	-	1	10	15	25-100
Construction	-	-	1	10	10-25

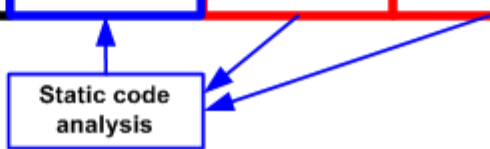


Figure 1 - Average costs of correcting defects depending upon the time of their appearance in the code and their detection (the data presented in the table are taken from the book 'Code Complete' by S. McConnell)

Static analysis tools reduce the cost of development of the whole project by detecting many errors at the stage of designing the code.

## Static analysis for detecting 64-bit errors

Let us point out the advantages of static code analysis that make this method the most appropriate to detect errors in 64-bit code:

1. You can check the **WHOLE** code. Analyzers can even test those code fragments that get control very seldom. In other words, static analyzers provide nearly full coverage of the code. It allows you to make sure that the whole code has been checked before you port it to a 64-bit system.
2. Scalability. Static analysis allows you to analyze both a small and a large project with equal simplicity. Labor intensiveness rises directly as the project size. You may easily distribute the project analysis among several developers. You need just to distribute the project's parts among the programmers.
3. While only beginning to work on a project, the developer will not fail to notice possible issues even without knowing all the peculiarities of the 64-bit code. The analyzer will point at the dangerous places and Help system will tell you everything you should know about this or that issue.
4. Costs are reduced due to early diagnosis of errors.
5. You may efficiently use static analysis tools both when porting code to a 64-bit system and developing a new 64-bit code.

## Viva64 analyzer included into PVS-Studio

[PVS-Studio](#) is a package of static code analyzers to check contemporary resource-intensive applications. PVS-Studio includes a special static analyzer Viva64 intended for diagnosing 64-bit errors.

PVS-Studio analyzer is designed for a Windows-platform. It integrates into Microsoft Visual Studio 2005/2008/2010 development environment (see Figure 2). PVS-Studio's interface allows you to filter diagnostic warnings using various techniques and also save and load warning lists.

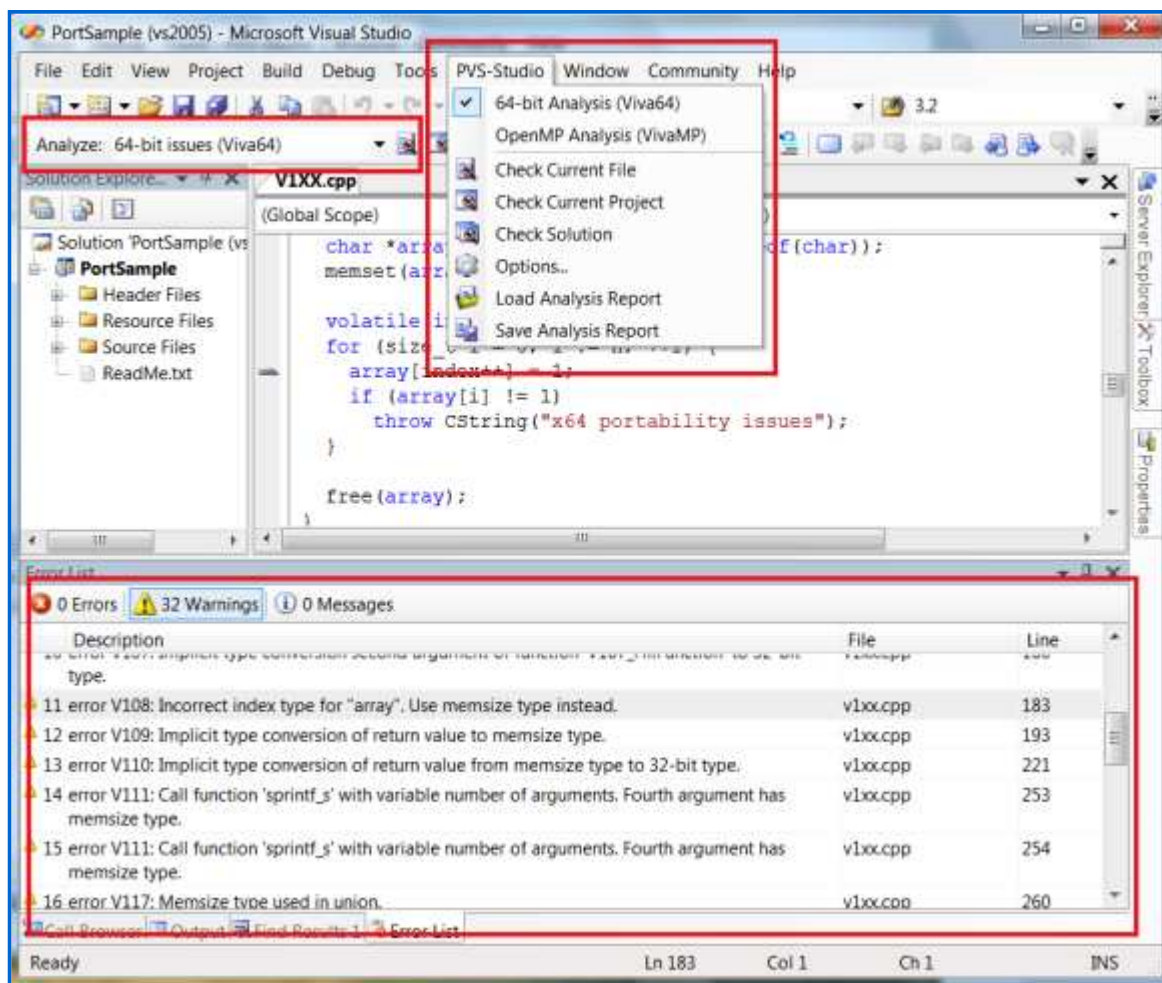


Figure 2 - PVS-Studio integrating into Microsoft Visual Studio

The analyzer's system requirements coincide with those of Microsoft Visual Studio:

- Operating system: Windows 7 / Windows 2000/XP/2003/Vista/2008 [x86](#) or [x64](#). Note that your operating system does not necessarily need to be a 64-bit one to enable you to analyze 64-bit applications.
- Development environment: Microsoft Visual Studio 2005/2008/2010 (Standard Edition, Professional Edition, Team Systems). You must have a Visual Studio component called "X64 Compilers and Tools" installed to be able to test 64-bit applications. It is included into all Visual Studio versions we have enumerated and can be installed through Visual Studio Setup. Note that PVS-Studio cannot work with Visual C++ Express because this system does not support add-in modules.
- Hardware: PVS-Studio can work on systems that have not less than 1 Gbyte of memory (it is recommended to have 2 Gbytes or more); the analyzer can work employing several cores (the more the cores, the faster the code analysis is).

All the errors that can be diagnosed are thoroughly described in Help system that integrates into MSDN and becomes available after you install PVS-Studio. You may also see [Help system on PVS-Studio](#) online on our site.

The PVS-Studio distribution kit also contains special projects serving as examples of code flaws that will help you study the analyzer.

You may download the [demo-version](#) and see the guide "[PVS-Studio Tutorial](#)" to get closer to PVS-Studio. The demo-version has several limitations:

- you may use it within 30 days;
- the demo-version hides the numbers of most lines containing errors and shows only some of them (although it detects all errors in the project). But we made an exception for the demonstration projects included into PVS-Studio - when you analyze these projects, you see all the numbers of the lines with defects.

At present, PVS-Studio analyzer provides the fullest diagnosis of 64-bit errors. You may study the comparison characteristics in the article "[Comparing PVS-Studio with other code analyzers](#)".

While studying various patterns of errors in the next lessons, we will often refer to PVS-Studio to show you how to detect them.

*The course authors: Andrey Karpov ([karpov@viva64.com](mailto:karpov@viva64.com)), Evgeniy Ryzhkov ([evg@viva64.com](mailto:evg@viva64.com)).*

*The rightholder of the course "Lessons on development of 64-bit C/C++ applications" is OOO "Program Verification Systems". The company develops software in the sphere of source program code analysis. The company's site: <http://www.viva64.com>.*

*Contacts: e-mail: [support@viva64.com](mailto:support@viva64.com), Tula, 300027, PO box 1800.*

## Lesson 9. Pattern 1. Magic numbers

In a poorly written code you may often see magic numeric constants whose presence is dangerous by itself. When porting code to a 64-bit platform, these constants may make the code inefficient if they participate in address computation, object size computation or bit operations.

Table 1 presents the basic magic constants that may impact efficiency of an application ported to a new platform.

Value:	Can be used as:
4	Number of bytes in type.
32	Number of bits in type.
0x7fffffff	Max value of signed variable. Mask for higher bit zero setting.
0x80000000	Min value of signed variable. Mask for higher bit selecting.
0xffffffff	Max value of unsigned variable. Alternative representation -1 as error indicator.

*Table 1 - The basic magic numbers which are dangerous when porting 32-bit applications to a 64-bit platform*

You should examine your code very attentively to check if it contains magic constants and replace them with safe constants and expressions. You may use the operator *sizeof()* or special values from *<limits.h>*, *<inttypes.h>*, etc. for that.

Here are examples of some errors related to magic constants. The most common error is writing type sizes in the form of numeric values:

```
1) size_t ArraySize = N * 4;
   intptr_t *Array = (intptr_t *)malloc(ArraySize);

2) size_t values[ARRAY_SIZE];
   memset(values, ARRAY_SIZE * 4, 0);

3) size_t n, r;
   n = n >> (32 - r);
```

In all these cases we assume that the size of the types used is always 4 bytes. To correct the code we should use the operator *sizeof()*:

```
1) size_t ArraySize = N * sizeof(intptr_t);
   intptr_t *Array = (intptr_t *)malloc(ArraySize);

2) size_t values[ARRAY_SIZE];
   memset(values, ARRAY_SIZE * sizeof(size_t), 0);
```

or

```
memset(values, sizeof(values), 0); //preferred alternative
```

```
3) size_t n, r;
   n = n >> (CHAR_BIT * sizeof(n) - r);
```

Sometimes you may need a specific constant. As an example, let us take the value of [size\\_t](#) where all the bytes except for the 4 lower bytes must be filled with ones. In a 32-bit program, this constant is defined in this way:

```
// constant '1111..110000'
const size_t M = 0xFFFFFFFF0u;
```

It is incorrect for a 64-bit system. Such errors are very unpleasant because magic constants may be written in various ways and it takes a lot of time and efforts to find them. Unfortunately, there are no other ways to find and correct such code fragments but to use the directive *#ifdef* or a special macro.

```
#ifdef _WIN64
#define CONST3264(a) (a##i64)
#else
#define CONST3264(a) (a)
#endif

const size_t M = ~CONST3264(0xFu);
```

Sometimes the value "-1" is used as an error code or other special marker and it is written as "0xffffffff". This expression is incorrect on a 64-bit platform, so you should explicitly define the value *-1*. Here is an example of incorrect code that uses the value 0xffffffff as an error marker:

```
#define INVALID_RESULT (0xFFFFFFFFu)

size_t MyStrLen(const char *str) {
    if (str == NULL)
        return INVALID_RESULT;
    ...
    return n;
}

size_t len = MyStrLen(str);
if (len == (size_t)(-1))
    ShowError();
```

To make it clear, let us explain what the value "(size\_t)(-1)" is equal to on a 64-bit platform. You will be mistaken saying it is 0x00000000FFFFFFFFFu. According to C++ rules, at first value *-1* is converted to a signed equivalent of a larger type and then to an unsigned value:

```
int a = -1;           // 0xFFFFFFFFi32
ptrdiff_t b = a;      // 0xFFFFFFFFFFFFFFFFi64
size_t c = size_t(b); // 0xFFFFFFFFFFFFFFFFui64
```

Thus, on a 64-bit platform, "(size\_t)(-1)" equals the value 0xFFFFFFFFFFFFFFFFFui64 which is the maximum value for the 64-bit *size\_t*.

Let us return to the error with INVALID\_RESULT. When 0xFFFFFFFFFu constant is used, the condition "len == (size\_t)(-1)" is not fulfilled in a 64-bit program. The best solution is to change the code so that it will not need special marker values. If you cannot refuse to use them due to some reason or do not want to significantly edit the code, simply use the explicit value *-1*.

```
#define INVALID_RESULT (size_t)(-1)
...
```

Here is one more example related to 0xFFFFFFFF. The code is taken from a real application of 3D modeling:

```
hFileMapping = CreateFileMapping (
    (HANDLE) 0xFFFFFFFF,
    NULL,
    PAGE_READWRITE,
    (DWORD) 0,
    (DWORD) (szBufIm),
    (LPCTSTR) &FileShareNameMap[0]);
```

As you have already guessed, 0xFFFFFFFF here also leads to an error on a 64-bit system. The first argument of the function *CreateFileMapping* may have the value INVALID\_HANDLE\_VALUE defined in this way:

```
#define INVALID_HANDLE_VALUE ((HANDLE)(LONG_PTR)-1)
```



As a result, `INVALID_HANDLE_VALUE` does coincide with the value `0xFFFFFFFF` on a 32-bit system. But on a 64-bit system, it is the value `0x00000000FFFFFFFF` which is passed into the function `CreateFileMapping`, so the system considers the argument incorrect and returns the code of the error. The cause is that the value `0xFFFFFFFF` has an `UNSIGNED` type (*unsigned int*). The value `0xFFFFFFFF` does not fit into the type *int* and therefore is *unsigned*. It is a subtle thing that you should consider when moving to 64-bit systems. Let us explain it by an example:

```
void foo(void *ptr)
{
    cout << ptr << endl;
}
int _tmain(int, _TCHAR *[])
{
    cout << "-1\t\t";
    foo((void *)-1);
    cout << "0xFFFFFFFF\t";
    foo((void *)0xFFFFFFFF);
}
```

The result of the 32-bit version of the program:

```
-1          FFFFFFFF
0xFFFFFFFF  FFFFFFFF
```

The result of the 64-bit version of the program:

```
-1          FFFFFFFFFFFFFFFFFF
0xFFFFFFFF  00000000FFFFFFFF
```

## Diagnosis

[PVS-Studio](#) static analyzer warns the programmer about magic constants present in code which are the most dangerous when developing a 64-bit application. The diagnostic warnings [V112](#) and [V118](#) are used for this purpose. Keep in mind that the analyzer does not warn you about a possible error if a magic constant is defined through a macro. For example:

```
#define MB_YESNO 0x00000004L
MessageBox("Are you sure ?", "Question", MB_YESNO);
```

In short, the reason for this behavior is false alarm protection. It supposes that when programmers define constants through macros, they do it consciously to emphasize that they are safe. To learn more about it see the blog-post on our site "[Magic constants and malloc\(\) function](#)".

*The course authors: Andrey Karpov ([karpov@viva64.com](mailto:karpov@viva64.com)), Evgeniy Ryzhkov ([evg@viva64.com](mailto:evg@viva64.com)).*

*The rightholder of the course "Lessons on development of 64-bit C/C++ applications" is OOO "Program Verification Systems". The company develops software in the sphere of source program code analysis. The company's site: <http://www.viva64.com>.*

*Contacts: e-mail: [support@viva64.com](mailto:support@viva64.com), Tula, 300027, PO box 1800.*

## Lesson 10. Pattern 2. Functions with variable number of arguments

Typical examples given in most articles on the issues of porting programs to 64-bit systems refer to incorrect use of the functions *printf*, *scanf* and their variants.

Example 1:

```
const char *invalidFormat = "%u";
size_t value = SIZE_MAX;
printf(invalidFormat, value);
```

Example 2:

```
char buf[9];
sprintf(buf, "%p", pointer);
```

In the first case, the programmer does not take into account that the type [size\\_t](#) is not equivalent to the type *unsigned* on a 64-bit platform. It will result in printing an incorrect result if `value > UINT_MAX`.

In the second case, the author does not take into account that the size of the pointer might be more than 32 bits in future. As a result, this code will cause a buffer overflow on the 64-bit architecture.

Incorrect use of functions with the variable number of arguments is a common error not only for 64-bit architectures but for all architectures. It is explained by the fundamental danger of using these C++ constructs. It is generally accepted to refuse using them and resort to safe programming methods. We highly recommend you to modify your code and employ safe methods. For example, you may replace *printf* with *cout*, *sprintf* with *boost::format* or *std::stringstream*.

This recommendation is often criticized by Linux developers who argue that gcc compiler checks if the format string corresponds to the actual arguments passed into the function *printf*. But they forget that the format string may be called from other program parts or loaded from resources. In other words, the format string is seldom present explicitly in the code of a real program and therefore the compiler cannot check it. If developers use Visual Studio 2005/2008 they will not be able to get the warning on the code like `"void *p = 0; printf("%x", p);"` even using the switches */W4* and */Wall*.

There exist size specifiers to work with memsize-types in functions like *sscanf*, *printf*. If you are developing a Windows application, you may use the "I" size specifier. For example:

```
size_t s = 1;
printf("%Iu", s);
```

If you are developing a Linux application, you may try the size specifier "z". For example:

```
size_t s = 1;
printf("%zu", s);
```

The specifiers are well described in the Wikipedia article "[printf](#)".

If you have to support a code being ported that uses functions like *sscanf*, you may employ special macros in the format of control strings that expand into the necessary size specifiers. Here is an example of a macro that helps create a portable code for various systems:

```
// PR_SIZET on Win64 = "I"
// PR_SIZET on Win32 = ""
// PR_SIZET on Linux64 = "z"
// ...
size_t u;
scanf("%" PR_SIZET "u", &u);
```

Here is one more example. Although it looks most strange, the code given here in an abridged form was used in a real application in the UNDO/REDO subsystem:

```
// Here the pointers were saved in the form of a string
int *p1, *p2;
....
char str[128];
sprintf(str, "%X %X", p1, p2);

// In another function this string was processed
// in this way:
void foo(char *str)
{
    int *p1, *p2;
    sscanf(str, "%X %X", &p1, &p2);
    // The result is incorrect values of pointers p1 and p2.
    ...
}
```

Manipulation with the pointers using "%X" resulted in an incorrect program behavior on a 64-bit system. This example shows how dangerous may be the depths of large and complex projects written for many years. If your project is rather large and obsolete, you might encounter very interesting fragments like this one.

## Diagnosis

Those types that change their sizes on a 64-bit system, i.e. [memsize-types](#), are dangerous for the functions with the variable number of arguments. [PVS-Studio](#) static analyzer warns the programmer about such types with the help of the [V111](#) diagnostic warning.

If the types of the arguments have not changed their sizes, the code is considered correct and no warnings are generated. Here is an example of code correct from the analyzer's viewpoint:

```
printf("%d", 10*5);
CString str;
size_t n = sizeof(float);
str.Format(StrFormat, static_cast<int>(n));
```

*The course authors: Andrey Karpov ([karpov@viva64.com](mailto:karpov@viva64.com)), Evgeniy Ryzhkov ([evg@viva64.com](mailto:evg@viva64.com)).*

*The rightholder of the course "Lessons on development of 64-bit C/C++ applications" is OOO "Program Verification Systems". The company develops software in the sphere of source program code analysis.*

The company's site: <http://www.viva64.com>.

Contacts: e-mail: [support@viva64.com](mailto:support@viva64.com), Tula, 300027, PO box 1800.

## Lesson 11. Pattern 3. Shift operations

It is easy to make a mistake in code that works with separate bits. The pattern of 64-bit errors under consideration relates to shift operations. Here is an example of code:

```
ptrdiff_t SetBitN(ptrdiff_t value, unsigned bitNum) {
    ptrdiff_t mask = 1 << bitNum;
    return value | mask;
}
```

This code works well on a 32-bit architecture and allows you to set a bit with numbers from 0 to 31 into one. After porting the program to a 64-bit platform you need to set bits from 0 to 63. But this code will never set the bits with the numbers 32-63. Note that the numerical literal "1" has *int* type and causes an overflow when a shift in 32 positions occurs as shown in Figure 1. As a result, we will get 0 (Figure 1-B) or 1 (Figure 1-C) depending on the compiler implementation.

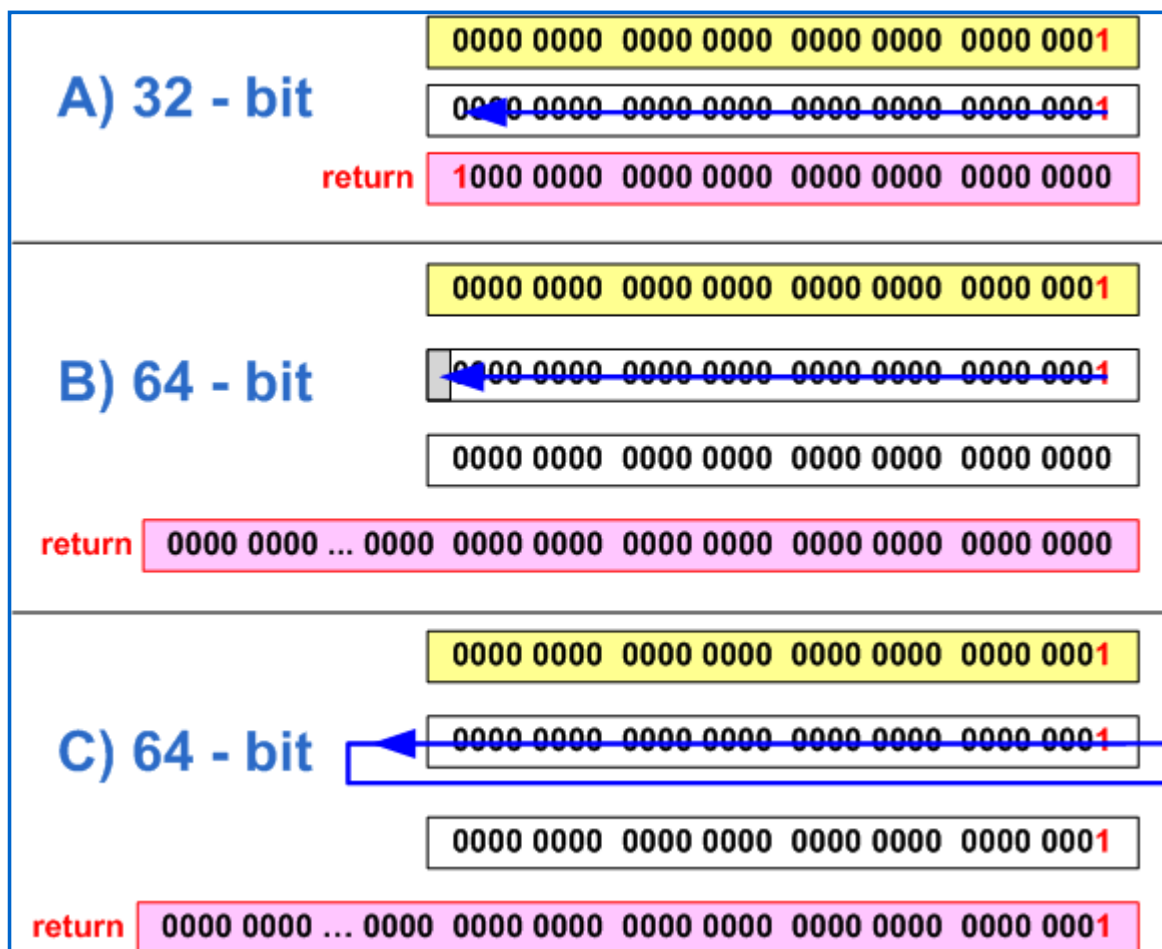


Figure 1 - a) Correct setting of the 31-st bit in a 32-bit code; b,c) - Incorrect setting of the 32-nd bit on a 64-bit system (two variants of behavior)

To correct the code we must make the type of the constant "1" the same as that of *mask* variable:

```
ptrdiff_t mask = ptrdiff_t(1) << bitNum;
```

Note also that the non-corrected code will lead to one more interesting error. When setting the 31-st bit on a 64-bit system, the function's result will be the value 0xffffffff80000000 (see Figure 2). The result of the expression  $1 \ll 31$  is the negative number -2147483648. This number is presented in a 64-bit integer variable as 0xffffffff80000000.

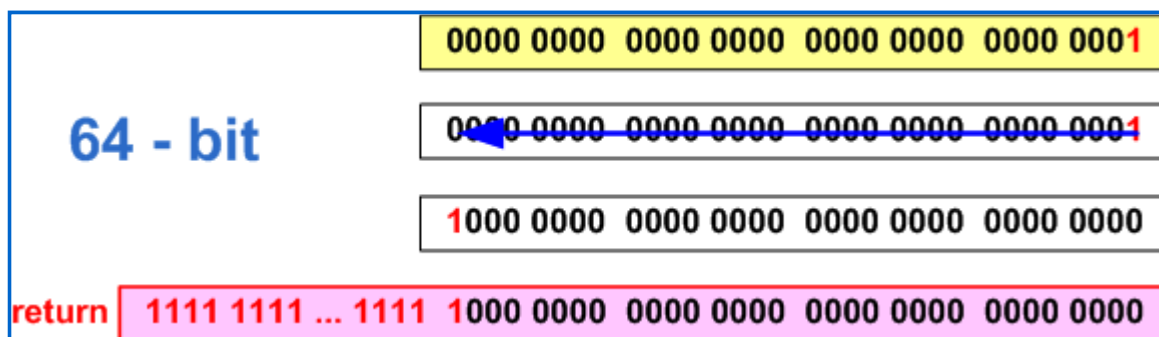


Figure 2 - The error of setting the 31-st bit on a 64-bit system.

You should remember and take into consideration the effects of shifting values of different types. To better understand all said above, consider some interesting expressions with shifts in a 64-bit system shown in Table 1.

Expression	Result (DEC)	Result (HEX)
<code>ptrdiff_t Result = 1 &lt;&lt; 31;</code>	-2147483648	0xffffffff80000000
<code>ptrdiff_t Result = 1U &lt;&lt; 31;</code>	2147483648	0x0000000080000000
<code>ptrdiff_t Result = ptrdiff_t(1) &lt;&lt; 31;</code>	2147483648	0x0000000080000000
<code>ptrdiff_t Result = 1 &lt;&lt; 32;</code>	0	0x0000000000000000
<code>ptrdiff_t Result = ptrdiff_t(1) &lt;&lt; 32;</code>	4294967296	0x0000000100000000

Table 1 - Expressions with shifts and their results in a 64-bit system (we used Visual C++ 2005 compiler)

The type of errors we have described is considered dangerous not only from the viewpoint of program operation correctness but from the viewpoint of security as well. Potentially, by manipulating with the input data of such incorrect functions one can get inadmissible rights when, for example, dealing with processing of access permissions' masks defined by separate bits. Questions related to exploiting errors in 64-bit code for application cracking and compromise are described in the article "[Safety of 64-bit code](http://www.viva64.com/lessons-x64/all_en.html?print=1)".

Now a subtler example:

```
struct BitFieldStruct {
    unsigned short a:15;
    unsigned short b:13;
};

BitFieldStruct obj;
obj.a = 0x4000;
size_t addr = obj.a << 17; //Sign Extension
printf("addr 0x%Ix\n", addr);

//Output on 32-bit system: 0x80000000
//Output on 64-bit system: 0xffffffff80000000
```

In the 32-bit environment, the order of calculating the expression will be as shown in Figure 3.

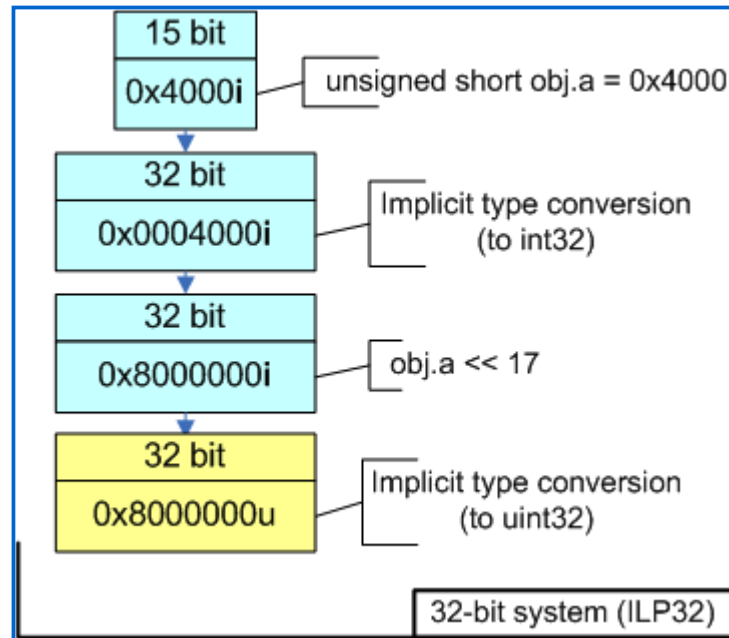


Figure 3 - Calculation of expression in 32-bit code

Note that a sign extension of "unsigned short" type to "signed int" takes place when calculating "obj.a << 17". To make it clear, consider the following code:

```
#include <stdio.h>

template <typename T> void PrintType(T)
{
    printf("type is %s %d-bit\n",
        (T)-1 < 0 ? "signed" : "unsigned", sizeof(T)*8);
}

struct BitFieldStruct {
    unsigned short a:15;
    unsigned short b:13;
};

int main(void)
{
    BitFieldStruct bf;
    PrintType( bf.a );
    PrintType( bf.a << 2);
    return 0;
}
```

Result:

```
type is unsigned 16-bit
type is signed 32-bit
```

Now let us see the consequence of the sign extension in a 64-bit code. The sequence of calculating the expression is shown in Figure 4.



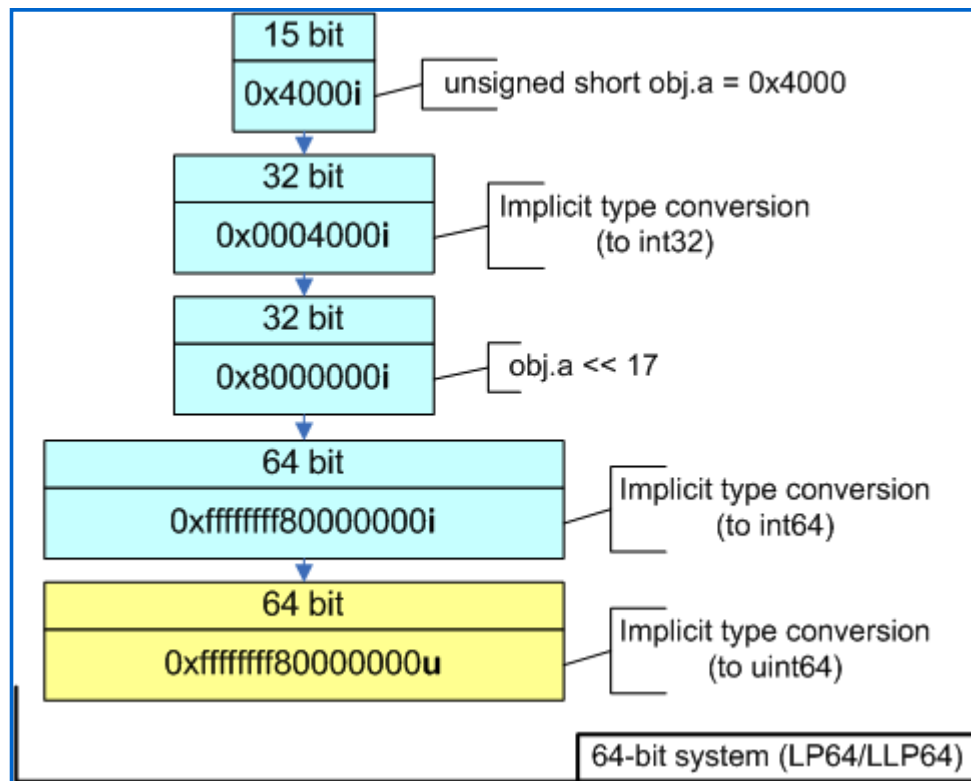


Figure 4 - Calculation of expression in 64-bit code

The member of "obj.a" structure is converted from the bit field of "unsigned short" type to "int". "obj.a << 17" expression has "int" type but it is converted to [ptrdiff\\_t](#) and then to [size\\_t](#) before it is assigned to addr variable. As a result, we will get the value 0xffffffff80000000 instead of 0x0000000080000000 expected.

Be careful when working with bit fields. To avoid the situation described in our example we need only to explicitly convert "obj.a" to size\_t type.

```

...
size_t addr = size_t(obj.a) << 17;
printf("addr 0x%Ix\n", addr);

//Output on 32-bit system: 0x80000000
//Output on 64-bit system: 0x80000000

```

## Diagnosis

Potentially unsafe shifts are detected by [PVS-Studio](#) static analyzer when it detects an implicit extension of a 32-bit type to [memsize](#) type. The analyzer will warn you about the unsafe construct with the diagnostic warning [V101](#). The shift operation is not suspicious by itself. But the analyzer detects an implicit extension of *int* type to *memsize* type when it is assigned to a variable, and informs the programmer about it to check the code fragment that may contain an error. Correspondingly, when there is no extension, the analyzer considers the code safe. For example: "int mask = 1 << bitNum;"

The course authors: Andrey Karpov ([karpov@viva64.com](mailto:karpov@viva64.com)), Evgeniy Ryzhkov ([evg@viva64.com](mailto:evg@viva64.com)).

The rightholder of the course "Lessons on development of 64-bit C/C++ applications" is OOO "Program

Verification Systems". The company develops software in the sphere of source program code analysis.  
The company's site: <http://www.viva64.com>.

Contacts: e-mail: [support@viva64.com](mailto:support@viva64.com), Tula, 300027, PO box 1800

## Lesson 12. Pattern 4. Virtual functions

Sometimes you may see errors there is nobody's fault about but they are still errors. Imagine that a long-long time ago (in Visual Studio 6.0) a project was developed that contained the class *CSampleApp* which was an heir of *CWinApp*. The base class had the function *WinHelp*. The heir overlapped this function and performed all the necessary actions. It looked as shown in Figure 1.

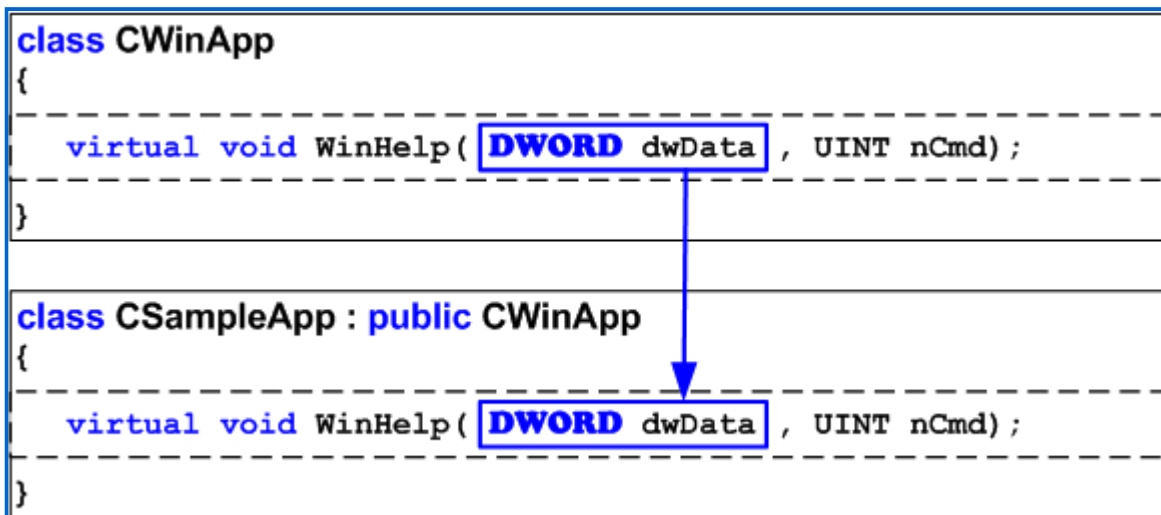


Figure 1 - Correct operable code created in Visual Studio 6.0

Then the project is ported to Visual Studio 2005 where the prototype of the function *WinHelp* has changed. But nobody notices it because the types *DWORD* and *DWORD\_PTR* coincide in the 32-bit mode and the program still works well (Figure 2).

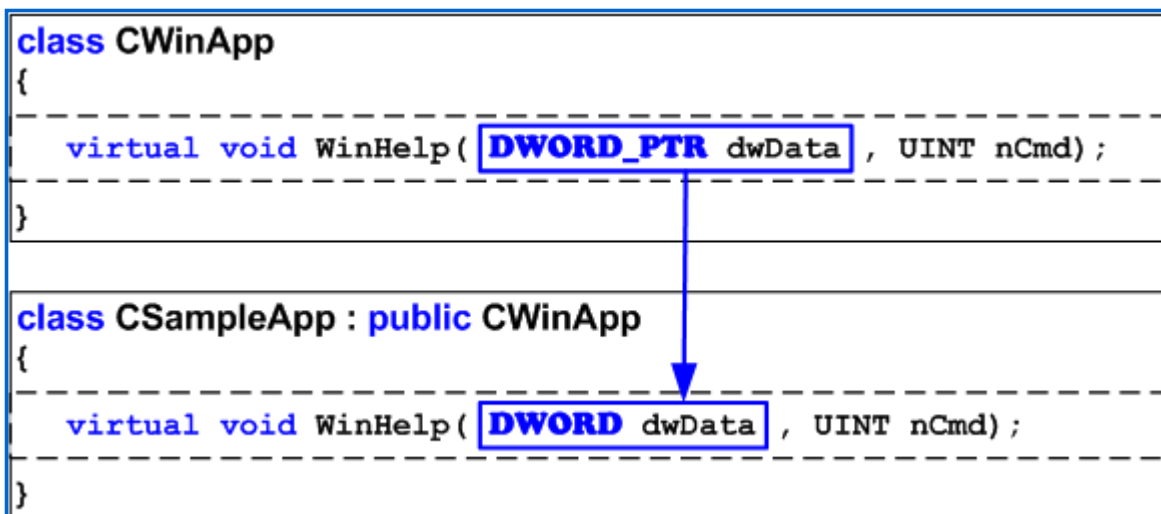


Figure 2 - Incorrect yet operable 32-bit code

The error waits to occur on a 64-bit system where the sizes of the types *DWORD* and *DWORD\_PTR*

differ (Figure 3). It turns out that the classes contain two DIFFERENT functions *WinHelp* in the 64-bit mode. Of course it is incorrect. Note that such traps may hide not only in MFC where some functions have different types of the arguments but in the code of your applications and third-party libraries as well.

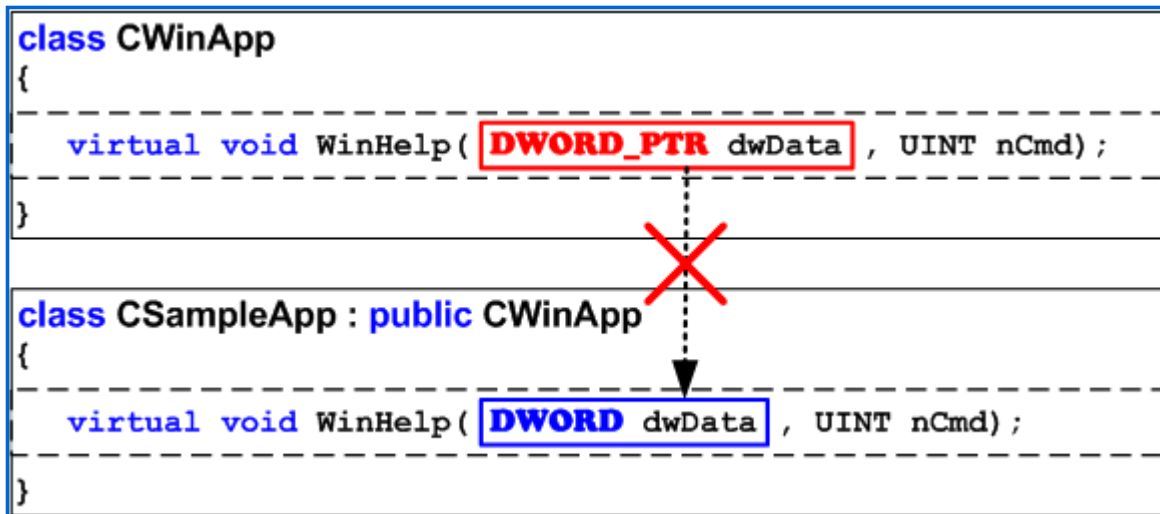


Figure 3 - The error occurs in the 64-bit code

Let us consider one more error by an example taken from real life. There is a wonderful component library BCGControlBar. You are likely to have heard about it because some components of BCGSoft Ltd company are included into Microsoft Visual Studio 2008 Feature Pack. Well, if you download the trial version of this library, install it and search for the word "WinHelp" through .h-files... you will see that wherever this function is supposedly overlapped the parameter DWORD is used instead of DWORD\_PTR. And it means that Help system will behave incorrectly in these classes when ported to a 64-bit system.

Why can such an error still exist in the code of so popular a library? We think the point is that the company's clients have access to the source codes of this library and they may always easily correct these codes. Besides, the function *WinHelp* is used very rarely nowadays. *HtmlHelp* is used much more frequently - and it does have the right parameter DWORD\_PTR in BCGControlBar. But the fact remains. There is an error in real code and the compiler does not detect it. Such errors may stay hidden for many years.

Note. This text is being written in December, 2009, and it is most likely that this error will be corrected in the next versions, especially as we have written about it to the developers of the library.

## Diagnosis

Errors related to virtual functions in 64-bit code can be detected by the static analyzer [PVS-Studio](#). The analyzer will warn you about dangerous virtual functions with the diagnostic warning [V301](#).

A virtual function is considered dangerous if:

1. The function is defined in the base class and in the heir-class.
2. The types of the functions' arguments do not coincide but are equivalent on a 32-bit system (for example: *unsigned*, *size\_t*) and are not equivalent on a 64-bit one.

The course authors: Andrey Karpov ([karpov@viva64.com](mailto:karpov@viva64.com)), Evgeniy Ryzhkov ([evg@viva64.com](mailto:evg@viva64.com)).

*The rightholder of the course "Lessons on development of 64-bit C/C++ applications" is OOO "Program Verification Systems". The company develops software in the sphere of source program code analysis. The company's site: <http://www.viva64.com>.*

*Contacts: e-mail: [support@viva64.com](mailto:support@viva64.com), Tula, 300027, PO box 1800.*

## Lesson 13. Pattern 5. Address arithmetic

We have chosen the 13-th lesson to discuss the errors related to [address arithmetic](#) deliberately. The errors related to pointer arithmetic in 64-bit systems are the most insidious and it would be good that number 13 made you more attentive.

The main idea of the pattern is - use only [memsize](#)-types in address arithmetic to avoid errors in 64-bit code.

Consider this code:

```
unsigned short a16, b16, c16;
char *pointer;
...
pointer += a16 * b16 * c16;
```

This sample works correctly with pointers if the result of the expression "a16 \* b16 \* c16" does not exceed INT\_MAX (2147483647). This code could always work correctly on a 32-bit platform, because on the 32-bit architecture a program does not have so much memory to create an array of such a size. On the 64-bit architecture, this limitation has been removed and the size of the array may well get larger than INT\_MAX items. Suppose we want to shift the value of the pointer in 6.000.000.000 bytes, so the variables *a16*, *b16* and *c16* have the values 3000, 2000 and 1000 respectively. When calculating the expression "a16 \* b16 \* c16", all the variables will be cast to "int" type at first, according to C++ rules, and only then they will be multiplied. An overflow will occur during the multiplication. The incorrect result will be extended to the type [ptrdiff\\_t](#) and the pointer will be calculated incorrectly.

You should be very attentive and avoid possible overflows when dealing with pointer arithmetic. It is good to use memsize-types or explicit type conversions in those expressions that contain pointers. Using an explicit type conversion we may rewrite our code sample in the following way:

```
short a16, b16, c16;
char *pointer;
...
pointer += static_cast<ptrdiff_t>(a16) *
           static_cast<ptrdiff_t>(b16) *
           static_cast<ptrdiff_t>(c16);
```

If you think that inaccurately written programs encounter troubles only when dealing with large data amounts, we have to disappoint you. Consider an interesting code sample working with an array that contains just 5 items. This code works in the 32-bit version and does not work in the 64-bit one:

```
int A = -2;
unsigned B = 1;
int array[5] = { 1, 2, 3, 4, 5 };
int *ptr = array + 3;
```

```
ptr = ptr + (A + B); //Invalid pointer value on 64-bit platform
printf("%i\n", *ptr); //Access violation on 64-bit platform
```

Let us follow the algorithm of calculating the expression "ptr + (A + B)":

- According to C++ rules, the variable A of the type *int* is converted to *unsigned*.
- A and B are summed and we get the value 0xFFFFFFFF of *unsigned* type.
- The expression "ptr + 0xFFFFFFFFu" is calculated.

The result of this process depends upon the size of the pointer on a particular architecture. If the addition takes place in the 32-bit program, the expression is equivalent to "ptr - 1" and the program successfully prints the value "3". In the 64-bit program, the value 0xFFFFFFFFu is fairly added to the pointer. As a result, the pointer gets far outside the array while we encounter some troubles when trying to get access to the item by this pointer.

Like in the first case, we recommend you to use only memsize-types in pointer arithmetic to avoid the situation described above. Here are two ways to correct the code:

```
ptr = ptr + (ptrdiff_t(A) + ptrdiff_t(B));

ptrdiff_t A = -2;
size_t B = 1;
...
ptr = ptr + (A + B);
```

You may argue and propose this way:

```
int A = -2;
int B = 1;
...
ptr = ptr + (A + B);
```

Yes, this code can work but it is bad due to some reasons:

- It trains programmers to be inaccurate when working with pointers. You might forget all the details of the code some time later and again redefine one of the variables with *unsigned* type by mistake.
- It is potentially dangerous to use non-memsize types together with the pointers. Suppose the variable Delta of *int* type participates in an expression with a pointer. This expression is quite correct. But an error may hide in the process of calculating the variable Delta because 32 bits might be not enough to perform the necessary calculations while working with large data arrays. You can automatically avoid this danger by using a memsize-type for the variable Delta.
- A code that uses the types *size\_t*, *ptrdiff\_t* and other memsize-types when working with pointers leads to a more appropriate binary code. We will speak about it in one of the following lessons.

## Array indexing

We single out this type of errors to make our description more structured because array indexing with the use of square brackets is just another way of writing the address arithmetic we have discussed above.

You may encounter errors related to indexing large arrays or eternal loops in programs that process large amounts of data. The following example contains 2 errors at once:

```

const size_t size = ...;
char *array = ...;
char *end = array + size;
for (unsigned i = 0; i != size; ++i)
{
    const int one = 1;
    end[-i - one] = 0;
}

```

The first error lies in the fact that an eternal loop may occur if the size of the processed data exceeds 4 Gbytes (0xFFFFFFFF), because the variable "i" has "unsigned" type and will never reach a value larger than 0xFFFFFFFF. It is possible but not certain - it depends upon the code the compiler will build. For example, there will be no eternal loop in the debug mode while it will completely disappear in the release version, because the compiler will decide to optimize the code using the 64-bit register for the counter and the loop will become correct. All this adds confusion and a code that was good yesterday stops working today.

The second error is related to negative values of the indexes serving to walk the array from end to beginning. This code works in the 32-bit mode but crashes in the 64-bit one right with the first iteration of the loop as an access outside the array's bounds occurs. Let us consider the cause of this behavior.

Although everything written below is the same as in the example with "ptr = ptr + (A + B)", we resort to this repetition deliberately. We need to show you that a danger may hide even in simple constructs and take various forms.

According to C++ rules, the expression "-i - one" will be calculated on a 32-bit system in the following way (i = 0 at the first step):

1. The expression "-i" has "unsigned" type and equals 0x00000000u.
2. The variable "one" is extended from the type "int" to the type "unsigned" and equals 0x00000001u. Note: the type "int" is extended (according to C++ standard) to the type "unsigned" if it participates in an operation where the second argument has the type "unsigned".
3. Two values of the type "unsigned" participate in a subtraction operation and its result equals 0x00000000u - 0x00000001u = 0xFFFFFFFFu. Note that the result has "unsigned" type.

On a 32-bit system, calling an array by the index 0xFFFFFFFFu is equivalent to using the index "-1". I.e. end[0xFFFFFFFFu] is analogous to end[-1]. As a result, the array's item is processed correctly. But the picture will be different in a 64-bit system: the type "unsigned" will be extended to the signed "ptrdiff\_t" and the array's index will equal 0x00000000FFFFFFFFFi64. It results in an overflow.

To correct the code you need to use such types as *ptrdiff\_t* and *size\_t*.

To completely convince you that you should use only memsize-types for indexing and in address arithmetic expressions, here is the code sample for you to consider.

```

class Region {
    float *array;
    int Width, Height, Depth;
    float Region::GetCell(int x, int y, int z) const;
    ...
};

float Region::GetCell(int x, int y, int z) const {

```



```

    return array[x + y * Width + z * Width * Height];
}

```

This code is taken from a real program of mathematical modeling where the amount of memory is the most important resource, so the capability of using more than 4 Gbytes on a 64-bit architecture significantly increases the computational power. Programmers often use one-dimensional arrays in programs like this to save memory while treating them as three-dimensional arrays. For this purpose, they use functions analogous to *GetCell* which provide access to the necessary items. But the code above will work correctly only with arrays that contain less than INT\_MAX items because it is 32-bit "int" types that are used to calculate the item's index.

Programmers often make a mistake trying to correct the code in this way:

```

float Region::GetCell(int x, int y, int z) const {
    return array[static_cast<ptrdiff_t>(x) + y * Width +
                z * Width * Height];
}

```

They know that, according to C++ rules, the expression to calculate the index has the type "ptrdiff\_t" and hope to avoid an overflow thereby. But the overflow may occur inside the expression "y \* Width" or "z \* Width \* Height" because it is still the type "int" which is used to calculate them.

If you want to correct the code without changing the types of the variables participating in the expression, you may explicitly convert each variable to a memsize-type:

```

float Region::GetCell(int x, int y, int z) const {
    return array[ptrdiff_t(x) +
                ptrdiff_t(y) * ptrdiff_t(Width) +
                ptrdiff_t(z) * ptrdiff_t(Width) *
                ptrdiff_t(Height)];
}

```

Another - better - solution is to change the types of the variables to a memsize-type:

```

typedef ptrdiff_t TCoord;
class Region {
    float *array;
    TCoord Width, Height, Depth;
    float Region::GetCell(TCoord x, TCoord y, TCoord z) const;
    ...
};

float Region::GetCell(TCoord x, TCoord y, TCoord z) const {
    return array[x + y * Width + z * Width * Height];
}

```

## Diagnosis

Address arithmetic errors are well diagnosed by [PVS-Studio](#) tool. The analyzer warns you about potentially dangerous expressions with the diagnostic warnings [V102](#) and [V108](#).

When possible, the analyzer tries to understand when a non-memsize type used in address arithmetic is

safe and refuse from generating a warning on this fragment. As a result, the analyzer's behavior may seem strange. In such cases we ask users to take their time and examine the situation. Consider the following code:

```
char Arr[] = { '0', '1', '2', '3', '4' };
char *p = Arr + 2;
cout << p[0u + 1] << endl;
cout << p[0u - 1] << endl; //V108
```

This code works correctly in the 32-bit mode and prints numbers 3 and 1 on the screen. On testing this code we get a warning only on one string with the expression "p[0u - 1]". And this warning is quite right! If you compile and launch this code sample in the 64-bit mode, you will see the value 3 printed on the screen and the program will crash right after it.

If you are sure that the indexing is correct, you may change the corresponding parameter of the analyzer on the settings tab [Settings: General](#) or use filters. You may also use an explicit type conversion.

*The course authors: Andrey Karpov ([karpov@viva64.com](mailto:karpov@viva64.com)), Evgeniy Ryzhkov ([evg@viva64.com](mailto:evg@viva64.com)).*

*The rightholder of the course "Lessons on development of 64-bit C/C++ applications" is OOO "Program Verification Systems". The company develops software in the sphere of source program code analysis. The company's site: <http://www.viva64.com>.*

*Contacts: e-mail: [support@viva64.com](mailto:support@viva64.com), Tula, 300027, PO box 1800.*

## Lesson 14. Pattern 6. Changing an array's type

Sometimes it is necessary (or simply convenient) to present array items in the form of items of another type. The following code example shows dangerous and safe type conversions:

```
int array[4] = { 1, 2, 3, 4 };
enum ENumbers { ZERO, ONE, TWO, THREE, FOUR };

//safe cast (for MSVC)
ENumbers *enumPtr = (ENumbers *) (array);
cout << enumPtr[1] << " ";

//unsafe cast
size_t *sizePtr = (size_t *) (array);
cout << sizePtr[1] << endl;

//Output on 32-bit system: 2 2
//Output on 64-bit system: 2 17179869187
```

As you may see, the output result of the program differs in the 32-bit and 64-bit versions. On the 32-bit system, the access to the array items is correct because the sizes of the types [size\\_t](#) and "int" coincide, so we see the result "2 2".

On the 64-bit system, the output result is "2 17179869187" because it is the value 17179869187 which is located in the 1-st item of the array *sizePtr* (see Figure 1). Sometimes it is this behavior you need but usually it is considered an error.

*Note. The type `enum` in Visual C++ compiler by default coincides with the type `int`, i.e. it is a 32-bit type. You may use `enum` of another size only with the help of an extension considered non-standard in Visual C++. So the example above is correct from the viewpoint of Visual C++ compiler but from the viewpoint of other compilers conversion of a pointer to `int` items to a pointer to `enum` items may be also incorrect.*

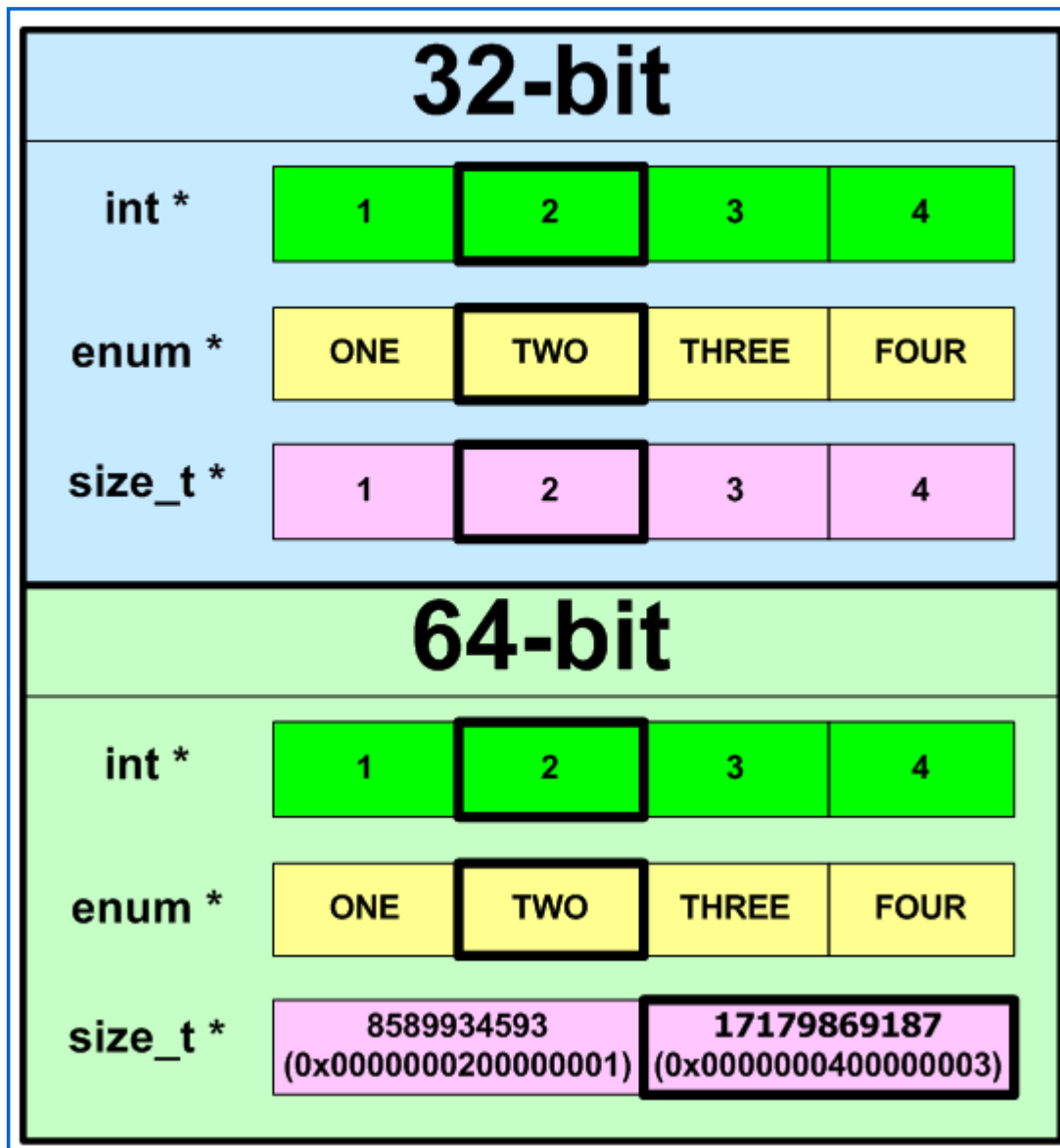


Figure 1 - Arrangement of array items in memory

To get rid of this incorrectness you should refuse to use unsafe type conversions and modify the program. Another way is to create a new array and copy the values from the original array into it.

You may encounter the described error pattern most often in the code fragments where programmers try to use pointer values as unique 32-bit identifiers.

## Diagnosis

Unsafe changes of an array's type are diagnosed by the tool [PVS-Studio](#). The analyzer warns you about potentially dangerous type conversions with the diagnostic warning [V114](#). Accordingly, the analyzer

responds only to those constructs that may cause an error on a 64-bit system. For example, the following code sample is correct and the analyzer will not call the programmer's attention to it:

```
void **pointersArray;
ptrdiff_t correctID = ((ptrdiff_t *)pointersArray)[index];
```

The course authors: Andrey Karpov ([karpov@viva64.com](mailto:karpov@viva64.com)), Evgeniy Ryzhkov ([evg@viva64.com](mailto:evg@viva64.com)).

The rightholder of the course "Lessons on development of 64-bit C/C++ applications" is OOO "Program Verification Systems". The company develops software in the sphere of source program code analysis. The company's site: <http://www.viva64.com>.

Contacts: e-mail: [support@viva64.com](mailto:support@viva64.com), Tula, 300027, PO box 1800.

## Lesson 15. Pattern 7. Pointer packing

A lot of errors occurring when porting code to 64-bit systems are related to changes of relations between the size of the pointer and the size of prime integers. In the environment with [ILP32](#) data model prime integers and pointers have the same size. Unfortunately, 32-bit code always relies on this assumption. Pointers are often cast to int, unsigned, long, DWORD and other inappropriate types.

You should always keep in mind that you must use only [memsize](#)-types for integer representation of pointers. We believe it is better to use the type [uintptr\\_t](#) because it reflects our intention better and makes the code more portable protecting it from changes in future.

Consider two small examples.

```
1) char *p;
   p = (char *) ((int)p & PAGEOFFSET);

2) DWORD tmp = (DWORD)malloc(ArraySize);
   ...
   int *ptr = (int *)tmp;
```

The both examples do not consider that the pointer's size might be other than 32 bits. Here an explicit type conversion is used that throws off the more significant bits of the pointer - this is an evident error on a 64-bit system. Below are the correct samples where integer memsize-types ([intptr\\_t](#) and [DWORD\\_PTR](#)) are used to pack the pointers:

```
1) char *p;
   p = (char *) ((intptr_t)p & PAGEOFFSET);

2) DWORD_PTR tmp = (DWORD_PTR)malloc(ArraySize);
   ...
   int *ptr = (int *)tmp;
```

The two examples discussed above are dangerous because the program failure might stay undetected for a long time. The program may work correctly when dealing with small data amounts on a 64-bit system as long as the processed addresses remain inside the first four Gbytes of memory. But then, as the program will be launched to solve large applied tasks, an overflow will occur outside this area. The code of the

examples will lead to an unpredictable program behavior when processing the pointer to an object situated outside this area.

The next code sample will reveal itself right away at the first run:

```
void GetBufferAddr(void **retPtr) {
    ...
    // Access violation on 64-bit system
    *retPtr = p;
}

unsigned bufAddress;
GetBufferAddr((void **)&bufAddress);
```

To correct it you should also use a type capable of storing the pointer.

```
size_t bufAddress;
GetBufferAddr((void **)&bufAddress); //OK
```

Sometimes it is just necessary to pack a pointer into a 32-bit type. It usually happens when you need to work with obsolete API functions. In these cases you should resort to special functions such as *LongToIntPtr*, *PtrToUlong*, etc.

To sum it up, I would like to note that it would be a bad style to pack a pointer into the types which always equal 64 bits. The code below will have to be corrected again when 128-bit systems appear:

```
PVOID p;
// Bad style. The 128-bit time will come.
__int64 n = __int64(p);
p = PVOID(n);
```

They say that Microsoft Research developers are already working on the task of providing compatibility of Windows 8 and Windows 9 cores with the 128-bit architecture. So, write a good code at once.

## Diagnosis

Packing of pointers into 32-bit types is diagnosed by the tool [PVS-Studio](#) with the diagnostic warnings [V114](#) and [V202](#).

We should note that simple errors related to conversions of pointers to 32-bit types are well diagnosed by Visual C++ compiler. For example, the compiler will warn you about the error in the code we have considered above:

```
char *p;
p = (char *) ((int)p & PAGEOFFSET);
```

Visual C++ will generate the warning "warning C4311: 'type cast': pointer truncation from 'char \*' to 'int'". But the example with *GetBufferAddr* will not be suspicious to Visual C++. So, PVS-Studio is more reliable than Visual C++ when you want to make sure that your code has no such errors.

Let us consider one more important thing related to PVS-Studio analyzer. Although some of the errors

can be detected with the help of Visual C++ warnings, it is sometimes impossible in practice. Many warnings are often disabled in large and obsolete projects (especially if they contain large third-party libraries) and therefore there is little probability that all 64-bit errors will be detected. On the other hand, it is not desirable to enable the warnings. There may be a lot of these but only some of them can really help you in detecting 64-bit errors. All the rest warnings will be irrelevant to the project and rather tell you about the inaccuracy of the code than potential 64-bit issues.

PVS-Studio is invaluable in this case. It will detect only those potential errors in the code which are directly related to the issues of porting the code to the 64-bit architecture.

*The course authors: Andrey Karpov ([karpov@viva64.com](mailto:karpov@viva64.com)), Evgeniy Ryzhkov ([evg@viva64.com](mailto:evg@viva64.com)).*

*The rightholder of the course "Lessons on development of 64-bit C/C++ applications" is OOO "Program Verification Systems". The company develops software in the sphere of source program code analysis. The company's site: <http://www.viva64.com>.*

*Contacts: e-mail: [support@viva64.com](mailto:support@viva64.com), Tula, 300027, PO box 1800.*

## Lesson 16. Pattern 8. Memsize-types in unions

A union is specific in that way that all the union items (members of the union) are assigned the same memory space, that is they are overlapped. Although you may access this memory space with the help of any member of the union, still you should choose it so that the result is sensible.

You should be very attentive dealing with unions that include pointers and other members of a [memsize](#)-type.

When you need to work with a pointer as an integer number, it may be convenient to use a union and work with the numerical representation of the type without explicit conversions. Consider the example:

```
union PtrNumUnion {
    char *m_p;
    unsigned m_n;
} u;

u.m_p = str;
u.m_n += delta;
```

This sample is correct for 32-bit systems and incorrect for 64-bit ones. Changing the member *m\_n* on a 64-bit system we work only with a part of the pointer *m\_p* (see Figure 1).



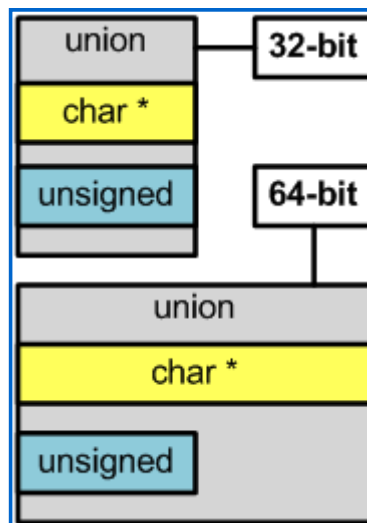


Figure 1 - The union format on the 32-bit and 64-bit systems

You should use a type that corresponds to the pointer's size:

```
union PtrNumUnion {
    char *m_p;
    size_t m_n; //type fixed
} u;
```

Another usual way of using a union is to represent one member as a set of several smaller members. For example, you may need to split a value of `size_t` type into bytes to implement the table algorithm of counting zero bits:

```
union SisetToBytesUnion {
    size_t value;
    struct {
        unsigned char b0, b1, b2, b3;
    } bytes;
} u;

SisetToBytesUnion u;
u.value = value;
size_t zeroBitsN = TranslateTable[u.bytes.b0] +
    TranslateTable[u.bytes.b1] +
    TranslateTable[u.bytes.b2] +
    TranslateTable[u.bytes.b3];
```

This code contains a fundamental algorithmic error that consists in the assumption that the type `size_t` contains 4 bytes. It is hardly possible at present to search for algorithmic errors in automatic mode but what we can do is to find all the unions and check if they contain memsize-types. On finding such a union we might encounter an error in it and rewrite the code in the following way.

```
union SisetToBytesUnion {
    size_t value;
    unsigned char bytes[sizeof(value)];
} u;

SisetToBytesUnion u;
u.value = value;
```

```
size_t zeroBitsN = 0;
for (size_t i = 0; i != sizeof(bytes); ++i)
    zeroBitsN += TranslateTable[bytes[i]];
```

## Diagnosis

The tool [PVS-Studio](#) allows the programmer to quickly find and look through all the unions that contain memsize-types in the program code. The analyzer generates the diagnostic warning [V117](#) for those structures the programmer should consider when porting the code to a 64-bit system.

*The course authors: Andrey Karpov ([karpov@viva64.com](mailto:karpov@viva64.com)), Evgeniy Ryzhkov ([evg@viva64.com](mailto:evg@viva64.com)).*

*The rightholder of the course "Lessons on development of 64-bit C/C++ applications" is OOO "Program Verification Systems". The company develops software in the sphere of source program code analysis. The company's site: <http://www.viva64.com>.*

*Contacts: e-mail: [support@viva64.com](mailto:support@viva64.com), Tula, 300027, PO box 1800.*

## Lesson 17. Pattern 9. Mixed arithmetic

I hope you have already rested from the 13-th lesson and now are ready to study one more important error pattern related to arithmetic expressions in which types of different capacities participate.

Mixed use of [memsize](#)-types and non-memsize types in expressions may lead to incorrect results on 64-bit systems and concern changes of the range of the input values. Consider some examples:

```
size_t Count = BigValue;
for (unsigned Index = 0; Index != Count; ++Index)
{ ... }
```

This is an example of an eternal loop occurring if `Count > UINT_MAX`. Suppose that this code works well on a 32-bit system with fewer iterations than `UINT_MAX`. But the 64-bit version of the program can process more data and may need more iterations. Since the values of `Index` variable lie within the range `[0..UINT_MAX]`, the condition "`Index != Count`" will never be fulfilled and it leads to the eternal loop.

*Note. Consider that this sample may work well at some particular settings of the compiler. Sometimes it is a source of much confusion because the code seems to be correct. In one of the following lessons we will tell you about phantom errors that reveal themselves only some time later. If you are already longing to learn why the code behaves so strangely, see the article ["A 64-bit horse that can count"](#).*

To correct the code you should use only memsize-types in the expressions. In our example we may change the type of the variable `Index` from "unsigned" to [size\\_t](#).

Another frequent error is using expressions of the following kind:

```
int x, y, z;
ptrdiff_t SizeValue = x * y * z;
```

We have already examined such examples with an arithmetic overflow that occurs when calculating

expressions using non-memsize types. The result was incorrect of course. The search and detection of this code fragment was complicated by the fact that compilers usually do not generate any warnings on it. From the viewpoint of C++ language it is an absolutely correct construct: several variables of "int" type are multiplied together, after that the result is implicitly extended to the type [ptrdiff\\_t](#) and is assigned to a variable.

Here is a small code sample that shows the danger of inaccurate expressions with mixed types (these results were obtained in Microsoft Visual C++ 2005 in the 64-bit compilation mode):

```
int x = 100000;
int y = 100000;
int z = 100000;
ptrdiff_t size = 1;           // Result:
ptrdiff_t v1 = x * y * z;     // -1530494976
ptrdiff_t v2 = ptrdiff_t (x) * y * z; // 1000000000000000
ptrdiff_t v3 = x * y * ptrdiff_t (z); // 141006540800000
ptrdiff_t v4 = size * x * y * z; // 1000000000000000
ptrdiff_t v5 = x * y * z * size; // -1530494976
ptrdiff_t v6 = size * (x * y * z); // -1530494976
ptrdiff_t v7 = size * (x * y) * z; // 141006540800000
ptrdiff_t v8 = ((size * x) * y) * z; // 1000000000000000
ptrdiff_t v9 = size * (x * (y * z)); // -1530494976
```

All the operands in such expressions must be cast to a type of a larger capacity while performing the calculations. Remember that an expression like

```
ptrdiff_t v2 = ptrdiff_t (x) + y * z;
```

does not guarantee a correct result at all. It guarantees only that the expression "ptrdiff\_t (x) + y \* z" will have the type "ptrdiff\_t".

So, if the expression's result must have a memsize-type, there must be only memsize-types in the expression too. Here is the correct version:

```
ptrdiff_t v2 = ptrdiff_t (x) + ptrdiff_t (y) * ptrdiff_t (z); // OK!
```

However, it is not always necessary to convert all the arguments to a memsize-type. If an expression consists of identical operators, you may convert only the first argument to the memsize-type. Consider an example:

```
int c();
int d();
int a, b;
ptrdiff_t v2 = ptrdiff_t (a) * b * c() * d();
```

The order of calculating the expression with the operators of the same priority has not been defined. More exactly, the compiler may choose any order of calculating the subexpressions (for example the calls of the functions c() and d()) it considers the most efficient, even if the subexpressions may cause side effects. The order of appearance of side effects has not been defined either. But since the multiplication operation refers to left-associative operators, the procedure of calculation will be performed in the following way:

```
ptrdiff_t v2 = ((ptrdiff_t (a) * b) * c()) * d();
```

As a result, each of the operators will be cast to the type "ptrdiff\_t" before the multiplication and we will get the correct result.

*Note. If there are integer calculations in your program and they need the control over overflows, resort to the class SafeInt - you may learn about its implementation and see its description in MSDN.*

Mixed use of types may also result in the changes in program logic:

```
ptrdiff_t val_1 = -1;
unsigned int val_2 = 1;
if (val_1 > val_2)
    printf ("val_1 is greater than val_2\n");
else
    printf ("val_1 is not greater than val_2\n");

//Output on 32-bit system: "val_1 is greater than val_2"
//Output on 64-bit system: "val_1 is not greater than val_2"
```

According to C++ rules, the variable *val\_1* is extended to the type "unsigned int" and becomes the value 0xFFFFFFFFu on a 32-bit system - the condition "0xFFFFFFFFu > 1" is fulfilled. On a 64-bit system, however, it is the variable *val\_2* that gets extended to the type "ptrdiff\_t" - in this case it is the expression "-1 > 1" which is checked. Figures 1 and 2 give the outlines of the transformations that take place.

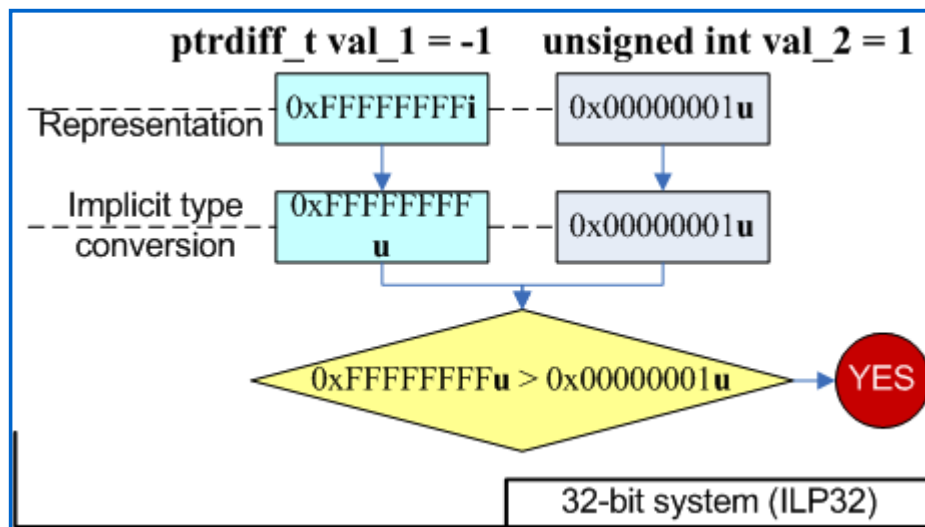


Figure 1 - Transformations taking place in the 32-bit version of the code

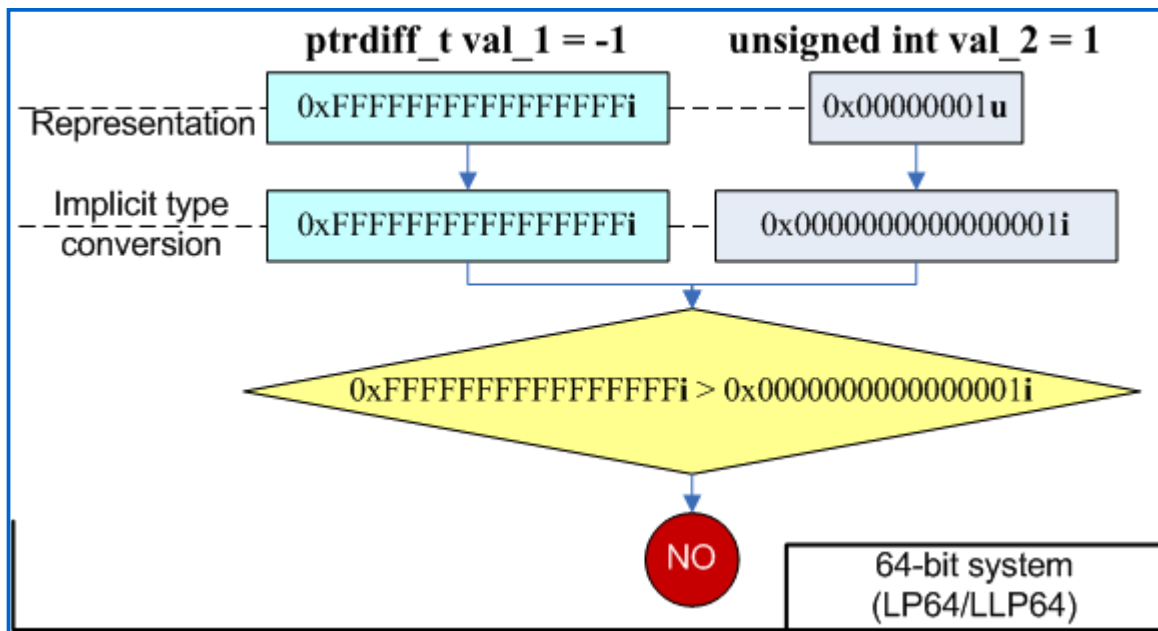


Figure 2 - Transformations taking place in the 64-bit version of the code

If you need to make the code behave in the same way as before, you should change the type of the variable `val_2`:

```
ptrdiff_t val_1 = -1;
size_t val_2 = 1;
if (val_1 > val_2)
    printf ("val_1 is greater than val_2\n");
else
    printf ("val_1 is not greater than val_2\n");
```

Actually, it would be more correct not to compare signed and unsigned types at all, but this issue lies beyond the current topic.

We have considered only simple expressions. But the described issues may occur when using other C++ constructs too:

```
extern int Width, Height, Depth;
size_t GetIndex(int x, int y, int z) {
    return x + y * Width + z * Width * Height;
}
...
MyArray[GetIndex(x, y, z)] = 0.0f;
```

If there is a large array (containing more than `INT_MAX` items), this code will be incorrect and we will be directed to the wrong items of the array `MyArray`. Although it is the value of "size\_t" type which is returned, the expression "`x + y * Width + z * Width * Height`" is calculated using the type "int". I think you have already guessed what the corrected code will look like:

```
extern int Width, Height, Depth;
size_t GetIndex(int x, int y, int z) {
    return (size_t)(x) +
        (size_t)(y) * (size_t)(Width) +
        (size_t)(z) * (size_t)(Width) * (size_t)(Height);
}
```

```
}
```

Or a bit simpler:

```
extern int Width, Height, Depth;
size_t GetIndex(int x, int y, int z) {
    return (size_t)(x) +
           (size_t)(y) * Width +
           (size_t)(z) * Width * Height;
}
```

In the next example again we have a mixture of a memsize-type (the pointer) and a 32-bit "unsigned" type:

```
extern char *begin, *end;
unsigned GetSize() {
    return end - begin;
}
```

The result of the expression "end - begin" has the type "ptrdiff\_t". Since the function returns the type "*unsigned*", there occurs an implicit type conversion that leads to a loss of the more significant bits of the result. So, if the pointers *begin* and *end* refer to the beginning and the end of the array whose size is more than UINT\_MAX (4Gb), the function will return an incorrect result.

And one more example. Here we are going to consider not a returned value but a formal argument of a function:

```
void foo(ptrdiff_t delta);

int i = -2;
unsigned k = 1;
foo(i + k);
```

This code resembles an example with incorrect [pointer arithmetic](#) discussed in the 13-th lesson, does not it? Right, here we have the same. We get the incorrect result when the actual argument, equaling 0xFFFFFFFF and having the type "unsigned", is implicitly extended to the type "ptrdiff\_t".

## Diagnosis

Errors occurring in 64-bit systems when integer types and memsize-types are used together are presented in many C++ syntactic constructs. To diagnose these errors several diagnostic warnings are used. PVS-Studio analyzer warns the programmer about possible errors with the help of these warnings: [V101](#), [V103](#), [V104](#), [V105](#), [V106](#), [V107](#), [V109](#), [V110](#), [V121](#).

Let us return to the example we have considered earlier:

```
int c();
int d();
int a, b;
ptrdiff_t x = ptrdiff_t(a) * b * c() * d();
```

Although the expression itself multiplies together the arguments extending their types to "ptrdiff\_t", an

error may hide in the procedure of calculating these arguments. That is why the analyzer still warns you about the mixed types: "V104: Implicit type conversion to memsize type in an arithmetic expression".

PVS-Studio tool also allows you to find potentially unsafe expressions which hide behind explicit type conversions. To enable this function you should enable the warnings [V201](#) and [V202](#). By default, the analyzer does not generate warnings concerning explicit type conversions. For example:

```
TCHAR *begin, *end;
unsigned size = static_cast<unsigned>(end - begin);
```

The warnings V201 and V202 will help you detect such incorrect code fragments.

Still the analyzer will pay no attention to type conversions which are safe from the viewpoint of the 64-bit code:

```
const int *constPtr;
int *ptr = const_cast<int>(constPtr);
float f = float(constPtr[0]);
char ch = static_cast<char>(sizeof(double));
```

*The course authors: Andrey Karpov ([karpov@viva64.com](mailto:karpov@viva64.com)), Evgeniy Ryzhkov ([evg@viva64.com](mailto:evg@viva64.com)).*

*The rightholder of the course "Lessons on development of 64-bit C/C++ applications" is OOO "Program Verification Systems". The company develops software in the sphere of source program code analysis.*

*The company's site: <http://www.viva64.com>.*

*Contacts: e-mail: [support@viva64.com](mailto:support@viva64.com), Tula, 300027, PO box 1800.*

## Lesson 18. Pattern 10. Storage of integer values in double

The type *double* has the capacity of 64 bits and is compatible with the standard IEEE-754 on 32-bit and 64-bit systems.

*Note. IEEE 754 is a widely spread standard of floating-point number presentation format used both in software and many hardware (CPU and FPU) implementations of arithmetic operations. Many compilers of programming languages use this standard to store and perform mathematical operations.*

Some programmers use the type *double* to store and work with integer types:

```
size_t a = size_t(-1);
double b = a;
--a;
--b;
size_t c = b; // x86: a == c
               // x64: a != c
```

This code may be justified when it is executed on a 32-bit system because the type *double* has 52 significant bits and can store a 32-bit integer value without loss. But when you save a 64-bit integer number into *double*, the exact result will be lost (see Figure 1).



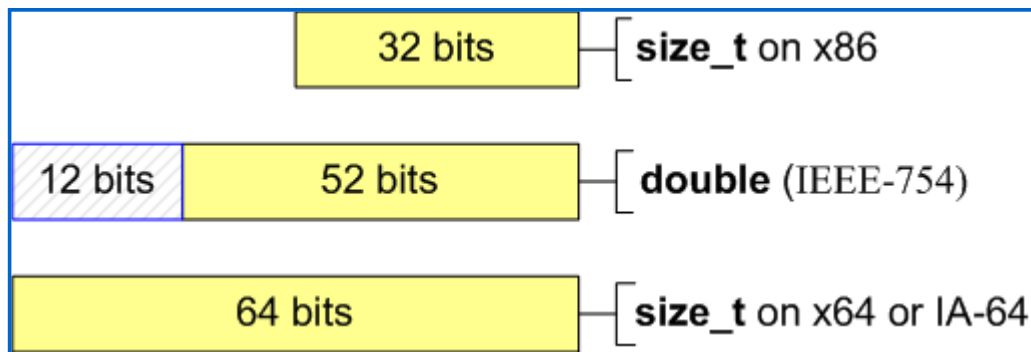


Figure 1 – The number of significant bits in the types `size_t` and `double`

Perhaps an approximate number will do in your program, but I would like to warn you just in case that you may encounter such consequences on the new architecture. And in no case would I advise you to mix integer arithmetic and floating-point arithmetic.

## Diagnosis

This error pattern is rather rare. However, these rare errors are in no way less dangerous. The analyzer PVS-Studio warns you about a potential error with the help of the diagnostic warning [V113](#). If you need to find explicit type conversions (from memsize-types to `double` and vice versa), you may enable the warning [V203](#).

The course authors: Andrey Karpov ([karpov@viva64.com](mailto:karpov@viva64.com)), Evgeniy Ryzhkov ([evg@viva64.com](mailto:evg@viva64.com)).

The rightholder of the course "Lessons on development of 64-bit C/C++ applications" is OOO "Program Verification Systems". The company develops software in the sphere of source program code analysis. The company's site: <http://www.viva64.com>.

Contacts: e-mail: [support@viva64.com](mailto:support@viva64.com), Tula, 300027, PO box 1800.

## Lesson 19. Pattern 11. Serialization and data interchange

Succession to existing data interchange protocols is an important component of the process of porting a program solution to a new platform. You need to provide the capability of reading the existing projects' formats, data interchange between 32-bit and 64-bit processes, etc.

Basically, the errors of this pattern concern serialization of [memsize](#)-types and data interchange operations they are used in:

```
1) size_t PixelsCount;
   fread(&PixelsCount, sizeof(PixelsCount), 1, inFile);

2) __int32 value_1;
   SSIZE_T value_2;
   inputStream >> value_1 >> value_2;
```

These samples contain errors of two kinds: using types of changeable capacity in binary interfaces and

ignoring the byte order.

## Using types of changeable capacity

Do not use types that change their sizes depending upon the development environment in binary interfaces of data interchange. All the types in C++ do not have a fixed size, so they cannot be used for this purpose. That is why developers of software development tools and programmers themselves create data types that have a strictly fixed size such as `__int8`, `__int16`, `INT32`, `word64`, etc.

These types enable data interchange between programs on various platforms although it requires some additional efforts. The two examples shown above are written incorrectly and it will become clear when some data types change their sizes from 32 bits to 64 bits. Keeping in mind the necessity of supporting obsolete data types, you may correct these samples in the following way:

```
1) size_t PixelsCount;
   __uint32 tmp;
   fread(&tmp, sizeof(tmp), 1, inFile);
   PixelsCount = static_cast<size_t>(tmp);

2) __int32 value_1;
   __int32 value_2;
   inputStream >> value_1 >> value_2;
```

But this way of correcting the code is not the best. The program may process more data after being ported to a 64-bit system, so 32-bit types in the code may become a great obstacle. In this case you may leave the obsolete code as it is to make it compatible with the obsolete data format but fix the incorrect types. Then you may create a new binary data format taking into consideration the previous errors. One more way out is to refuse to use binary formats and take the text format or other formats provided by various libraries.

## Ignoring the byte order

Even after solving the issue of type sizes, you may encounter the problem of incompatibility of binary formats. The cause lies in a different data representation. It is most often related to a different byte order.

A byte order is a method of writing the bytes of multibyte numbers (see also Figure 1). The little-endian byte order means that the writing begins with the least-significant byte and ends with the most-significant byte. This writing order is employed in the memory of personal computers with [x86](#) and [x86-64](#)-processors. The big-endian byte order means that the writing begins with the most-significant byte and ends with the least-significant byte. This order is a standard for TCP/IP protocols. That is why the big-endian byte order is often called the network byte order. This byte order is used in Motorola 68000 and SPARC processors.

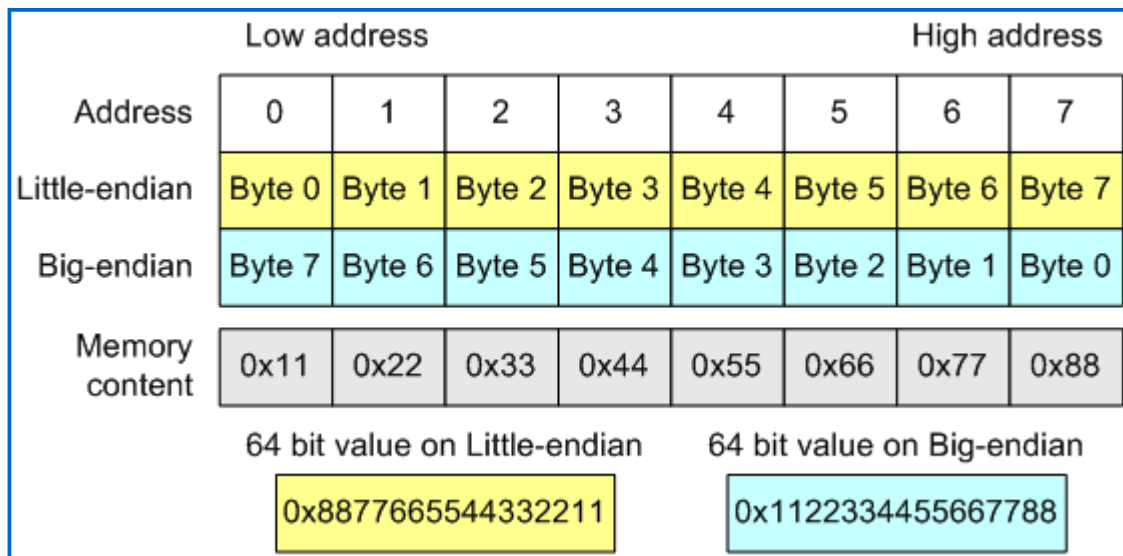


Figure 1 - The byte order in a 64-bit type in little-endian and big-endian systems

While developing a binary interface or data format, you should remember about the byte order. If the 64-bit system you are porting your 32-bit application to has a byte order different from that of your application, you will have to adjust your code to this difference. To convert the big-endian byte order into the little-endian byte order and vice versa you may use the functions `htonl()`, `htons()`, `bswap_64`, etc.

*Note. Many systems lack functions like `bswap_64` while the function `ntohl()` allows you to reverse only 32-bit values. They forgot to add a version of this function for 64-bit types for some reason. If you need to change the byte order in a 64-bit variable, see the discussion of the topic "[64 bit ntohl\(\) in C++ ?](#)" on [stackoverflow.com](#) site - there are several examples of how to implement this function.*

## Diagnosis

Unfortunately, [PVS-Studio](#) does not provide diagnosis for this pattern of 64-bit errors because this process cannot be formalized (we cannot compose a diagnostic rule). The only thing we may recommend you is to look through all the code fragments that are responsible for writing and reading data as well as sending data into other processes through the COM technology, for instance.

We would be glad if somebody of our readers proposed some ideas of how to detect the errors of this kind, at least partly.

The course authors: Andrey Karpov ([karpov@viva64.com](mailto:karpov@viva64.com)), Evgeniy Ryzhkov ([evg@viva64.com](mailto:evg@viva64.com)).

The rightholder of the course "Lessons on development of 64-bit C/C++ applications" is OOO "Program Verification Systems". The company develops software in the sphere of source program code analysis. The company's site: <http://www.viva64.com>.

Contacts: e-mail: [support@viva64.com](mailto:support@viva64.com), Tula, 300027, PO box 1800.

## Lesson 20. Pattern 12. Exceptions

Generation and processing of exceptions using integer types is a bad practice of C++ programming. You should use more informative types for these purposes, for example types derived from the class

`std::exception`. But sometimes you have to deal with a low-quality code like this:

```
char *ptr1;
char *ptr2;
try {
    try {
        throw ptr2 - ptr1;
    }
    catch (int) {
        std::cout << "catch 1: on x86" << std::endl;
    }
}
catch (ptrdiff_t) {
    std::cout << "catch 2: on x64" << std::endl;
}
```

You should be very attentive and avoid generation or processing of exceptions using [memsize](#)-types because it may result in changes of program logic. To correct this code you may replace "catch (int)" with "catch (ptrdiff\_t)". A more correct way is to use a special class to pass the information about an error that has occurred.

## Diagnosis

We have not encountered errors of this type in practice yet but the tool [PVS-Studio](#) can detect them. The diagnostic message [V115](#) will be shown when an exception is generated with the help of a memsize-type, while the warning [V116](#) will be generated when a memsize-type is used in *catch* operator.

*The course authors: Andrey Karpov ([karpov@viva64.com](mailto:karpov@viva64.com)), Evgeniy Ryzhkov ([evg@viva64.com](mailto:evg@viva64.com)).*

*The rightholder of the course "Lessons on development of 64-bit C/C++ applications" is OOO "Program Verification Systems". The company develops software in the sphere of source program code analysis.*

*The company's site: <http://www.viva64.com>.*

*Contacts: e-mail: [support@viva64.com](mailto:support@viva64.com), Tula, 300027, PO box 1800.*

## Lesson 21. Pattern 13. Data alignment

Processors work more efficiently when the data are aligned properly and some processors cannot work with non-aligned data at all. When you try to work with non-aligned data on [IA-64](#) (Itanium) processors, it will lead to generating an exception, as shown in the following example:

```
#pragma pack (1) // Also set by key /Zp in MSVC
struct AlignSample {
    unsigned size;
    void *pointer;
} object;

void foo(void *p) {
    object.pointer = p; // Alignment fault
}
```

If you have to work with non-aligned data on Itanium, you should specify this explicitly to the compiler.

For example, you may use a special macro UNALIGNED:

```
#pragma pack (1) // Also set by key /Zp in MSVC
struct AlignSample {
    unsigned size;
    void *pointer;
} object;

void foo(void *p) {
    *(UNALIGNED void *)&object.pointer = p; //Very slow
}
```

In this case the compiler generates a special code to deal with the non-aligned data. It is not very efficient since the access to the data will be several times slower. If your purpose is to make the structure's size smaller, you can get the best result arranging the data in decreasing order of their sizes. We will speak about it in more detail in one of the next lessons.

Exceptions are not generated when you address non-aligned data on the architecture [x64](#) but you still should avoid them - first, because the access to these data is very much slower, and second, because you may want to port the program to the platform IA-64 in the future.

Consider one more code sample that does not consider the data alignment:

```
struct MyPointersArray {
    DWORD m_n;
    PVOID m_arr[1];
} object;
...
malloc( sizeof(DWORD) + 5 * sizeof(PVOID) );
...
```

If we want to allocate an amount of memory needed to store an object of *MyPointersArray* type that contains 5 pointers, we should consider that the beginning of the array *m\_arr* will be aligned on an 8-byte boundary. The arrangement of data in memory in various systems (Win32/Win64) is shown in Figure 1.

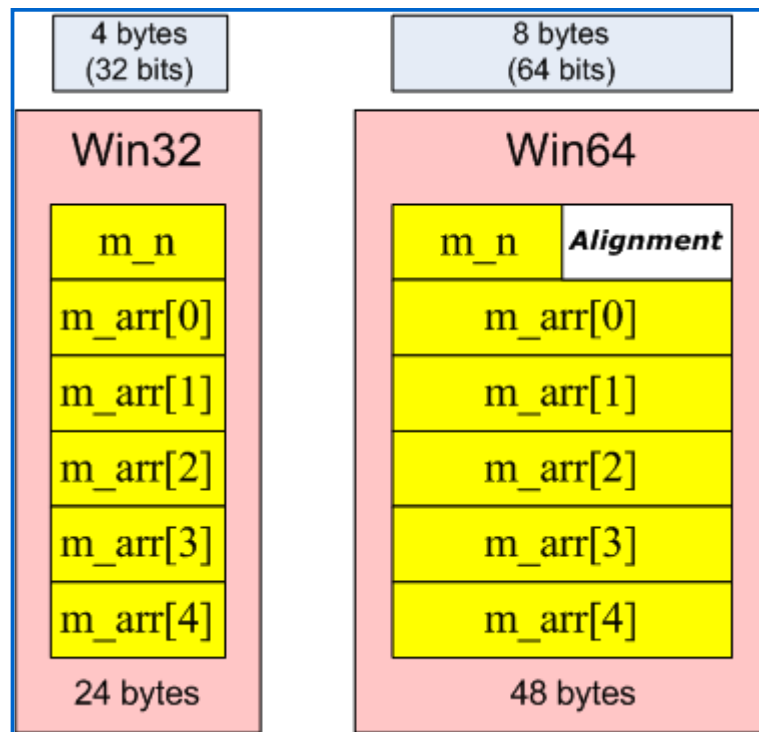


Figure 1- Data alignment in memory in Win32 and Win64 systems

The correct calculation of the size looks as follows:

```
struct MyPointersArray {
    DWORD m_n;
    PVOID m_arr[1];
} object;
...
malloc( FIELD_OFFSET(struct MyPointersArray, m_arr) +
        5 * sizeof(PVOID) );
...
```

In this code we find out the offset of the structure's last member and add this value to its size. You can find out the offset of a structure's or class's member with the help of the macro "offsetof" or FIELD\_OFFSET.

Always use these macros to know the offset in the structure without relying on knowing the types' sizes and alignment. Here is an example of code where the address of a structure's member is calculated correctly:

```
struct TFoo {
    DWORD_PTR whatever;
    int value;
} object;

int *valuePtr =
    (int *)((size_t)(&object) + offsetof(TFoo, value)); // OK
```

Linux-developers may encounter one more trouble related to alignment. You may learn what it is from our blog-post "[Change of type alignment and the consequences](http://www.viva64.com/lessons-x64/all_en.html?print=1)".

## Diagnosis

Since work with non-aligned data does not cause an error on the x64 architecture and only reduces performance, the tool [PVS-Studio](#) does not warn you about packed structures. But if the performance of an application is crucial to you, we recommend you to look through all the fragments in the program where "#pragma pack" is used. This is more relevant for the architecture IA-64 but PVS-Studio analyzer is not designed to verify programs for IA-64 yet. If you deal with Itanium-based systems and are planning to purchase PVS-Studio, write to us and we will discuss the issues of adapting our tool to IA-64 specifics.

PVS-Studio tool allows you to find errors related to calculation of objects' sizes and offsets. The analyzer detects dangerous arithmetic expressions containing several operators *sizeof()* (it signals a potential error). The number of the corresponding diagnostic message is [V119](#).

However, it is correct in many cases to use several *sizeof()* operators in one expression and the analyzer ignores such constructs. Here is an example of safe expressions with several *sizeof* operators:

```
int MyArray[] = { 1, 2, 3 };
size_t MyArraySize =
    sizeof(MyArray) / sizeof(MyArray[0]); //OK
assert(sizeof(unsigned) < sizeof(size_t)); //OK
size_t strLen = sizeof(String) - sizeof(TCHAR); //OK
```

## Appendix

Figure 2 represents types' sizes and their alignment. To learn about objects' sizes and their alignment on various platforms, see the code sample given in the blog-post "[Change of type alignment and the consequences](#)".



Type	32-bit x86 Windows		64-bit x86-64 Windows		32-bit x86 GNU/Linux		64-bit x86-64 GNU/Linux	
	sizeof	alignof	sizeof	alignof	sizeof	alignof	sizeof	alignof
<b>bool</b>	1	1	1	1	1	1	1	1
<b>wchar_t</b>	2	2	2	2	4	4	4	4
<b>short int</b>	2	2	2	2	2	2	2	2
<b>int</b>	4	4	4	4	4	4	4	4
<b>long int</b>	4	4	4	4	4	4	8	8
<b>long long int</b>	8	8	8	8	8	4	8	8
<b>float</b>	4	4	4	4	4	4	4	4
<b>double</b>	8	8	8	8	8	4	8	8
<b>long double</b>	8	8	8	8	12	4	16	16
<b>void*</b>	4	4	8	8	4	4	8	8

Figure 2 - Types' sizes and their alignment.

The course authors: Andrey Karpov ([karpov@viva64.com](mailto:karpov@viva64.com)), Evgeniy Ryzhkov ([evg@viva64.com](mailto:evg@viva64.com)).

The rightholder of the course "Lessons on development of 64-bit C/C++ applications" is OOO "Program Verification Systems". The company develops software in the sphere of source program code analysis. The company's site: <http://www.viva64.com>.

Contacts: e-mail: [support@viva64.com](mailto:support@viva64.com), Tula, 300027, PO box 1800.

## Lesson 22. Pattern 14. Overloaded functions

When porting a 32-bit program to a 64-bit platform, you may encounter changes in its logic related to the use of overloaded functions. If a function is overlapped for 32-bit and 64-bit values, the access to it with an argument of a [memsize](#)-type will be translated into different calls on different systems. This technique may be useful as, for example, in this code:

```
static size_t GetBitCount(const unsigned __int32 &) {
    return 32;
}

static size_t GetBitCount(const unsigned __int64 &) {
    return 64;
}

size_t a;
size_t bitCount = GetBitCount(a);
```

But this change of logic is potentially dangerous. Imagine a program that uses a class to arrange the stack. This class is specific in that way that it allows you to store values of different types:

```
class MyStack {
...
public:
    void Push(__int32 &);
    void Push(__int64 &);
    void Pop(__int32 &);
    void Pop(__int64 &);
} stack;

ptrdiff_t value_1;
stack.Push(value_1);
...
int value_2;
stack.Pop(value_2);
```

A careless programmer saves into and then selects from the stack values of different types ("[ptrdiff\\_t](#)" and "int"). Their sizes coincide on the 32-bit system and everything is quite okay. But when the size of the type "ptrdiff\_t" changes on the 64-bit system, the number of bytes saved into the stack gets larger than the number of bytes loaded then from the stack.

I think this type of errors is clear to you and you understand that one should be very careful about calls to overloaded functions when passing actual arguments of a memsize-type.

### Diagnosis

[PVS-Studio](#) does not diagnose this pattern of 64-bit errors. First, it is explained by the fact that we have not encountered such an error in a real application yet, and second, diagnosis of such constructs involves some difficulties. Please write to us if you encounter such an error in real code.

*The course authors: Andrey Karpov ([karpov@viva64.com](mailto:karpov@viva64.com)), Evgeniy Ryzhkov ([evg@viva64.com](mailto:evg@viva64.com)).*

The rightholder of the course "Lessons on development of 64-bit C/C++ applications" is OOO "Program Verification Systems". The company develops software in the sphere of source program code analysis. The company's site: <http://www.viva64.com>.

Contacts: e-mail: [support@viva64.com](mailto:support@viva64.com), Tula, 300027, PO box 1800.

## Lesson 23. Pattern 15. Growth of structures' sizes

A growth of structures' sizes is not an error by itself but it may lead to consumption of an unreasonably large memory amount and therefore to performance penalty. Let us consider this pattern not as an error but as a cause of 64-bit code inefficiency.

Data in structures of C++ language are aligned in such a way as to make the access to them most effective. Some microprocessors cannot address non-aligned data at all and the compiler has to generate a special code to deal with them. Those microprocessors that can address non-aligned data do it much less efficiently. That is why the C++ compiler leaves empty locations between structures' fields to align them on the addresses of machine words and therefore speed up the access to them. You may disable alignment using special `#pragma` directives to reduce the amount of memory being consumed but we are not interested in this way now. The amount of memory being used may often be greatly reduced by simply changing the order of fields in the structure without performance penalty.

Consider the following structure:

```
struct MyStruct
{
    bool m_bool;
    char *m_pointer;
    int m_int;
};
```

This structure will take 12 bytes on a 32-bit system and we cannot make it less. Each field is aligned on a 4-byte boundary. Even if we move `m_bool` to the end, it will not change anything. The compiler will still make the structure's size multiple of 4 bytes to align such structures in arrays.

In the 64-bit build mode the structure `MyStruct` will take 24 bytes. It is clear. First there is one byte for `m_bool` and 7 vacant bytes for the purpose of alignment because a pointer takes 8 bytes and must be aligned on an 8-byte boundary. Then there are 4 bytes for `m_int` and 4 vacant bytes to align the structure on an 8-byte boundary.

Fortunately, we may easily fix it by moving `m_bool` in the end of the structure, as shown below:

```
struct MyStructOpt
{
    char *m_pointer;
    int m_int;
    bool m_bool;
};
```

The structure `MyStructOpt` takes 16 bytes instead of 24. The arrangement of the fields is represented in Figure 1. It is rather a great saving if we use, for instance, 10 million items. In this case we will save 80 Mbytes of memory but what is more significant, we will enhance performance. If there will be few

structures, their sizes will not matter - the access will be performed with the same speed. But when there are many items, such things as cash, the number of memory accesses, etc. become significant. And you may say with certainty that 160 Mbytes of data will take less time to process than 240 Mbytes. Even a simple access to all the array items for reading will be faster.

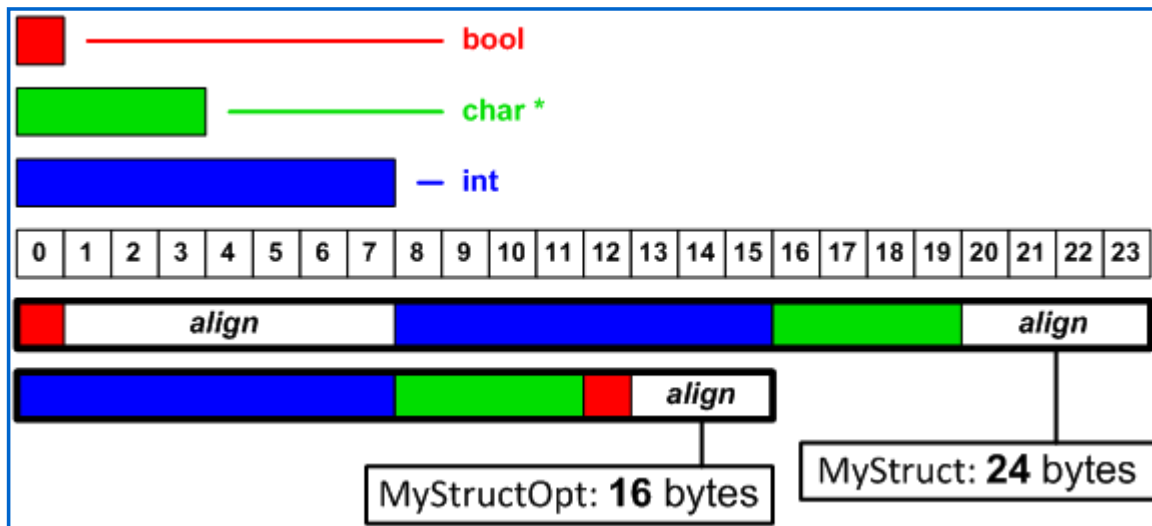


Figure 1 - Arrangement of the fields in the structures *MyStruct* and *MyStructOpt*

It is not always possible or convenient to change the order of fields in structures. But if there are millions of such structures, you must find some time for refactoring. The result of such simple optimization as changing the field order may be very great.

You may ask according to what rules the compiler aligns the data. We will answer briefly, but if you want to study this issue in more detail, read the book by Jeffery Richter "Programming Applications for MS Windows". This question is considered rather thoroughly there.

In general, the alignment rule is as follows: each field is aligned on the address multiple of the size of this field. A field of *size\_t* type on a 64-bit system will be aligned on an 8-byte boundary, *int* on a 4-byte boundary, *short* on a 2-byte boundary. Fields of *char* type are not aligned. The size of such a structure is aligned on the size multiple of the size of its maximum item. Let us explain this type of alignment by an example:

```
struct ABCD
{
    size_t m_a;
    char m_b;
};
```

The items will take  $8 + 1 = 9$  bytes. But if we want to create an array of structures `ABCD[2]`, the size of the structure being 9 bytes, the field `m_a` of the second structure will lie on the non-aligned address. Therefore the compiler will add 7 empty bytes to the structure to make its size 16 bytes.

The process of optimizing a field arrangement may seem complicated. But there is a very simple and very effective method: you just need to arrange the fields in decreasing order of their sizes. This will be quite enough. In this case, the fields will be arranged without unnecessary gaps. For example, take the following structure of 40 bytes:

```
struct MyStruct
{
    int m_int;
    size_t m_size_t;
    short m_short;
    void *m_ptr;
    char m_char;
};
```

By simply sorting the sequence of the fields in decreasing order of their sizes:

```
struct MyStructOpt
{
    void *m_ptr;
    size_t m_size_t;
    int m_int;
    short m_short;
    char m_char;
};
```

we make this structure's size only 24 bytes.

## Diagnosis

The tool [PVS-Studio](#) allows you to find structures in the code of 64-bit applications, whose sizes may be reduced by rearranging the fields in them. The analyzer generates the diagnostic message [V401](#) on non-optimal structures.

The analyzer does not always generate a warning about inefficient structures because it tries to avoid too many unnecessary warnings. For example, the analyzer does not generate a message on complex heir classes because such objects are usually very few. For example:

```
class MyWindow : public CWnd {
    bool m_isActive;
    size_t m_sizeX, m_sizeY;
    char m_color[3];
    ...
};
```

You may reduce this structure's size but there is no practical sense in it.

*The course authors: Andrey Karpov ([karpov@viva64.com](mailto:karpov@viva64.com)), Evgeniy Ryzhkov ([evg@viva64.com](mailto:evg@viva64.com)).*

*The rightholder of the course "Lessons on development of 64-bit C/C++ applications" is OOO "Program Verification Systems". The company develops software in the sphere of source program code analysis. The company's site: <http://www.viva64.com>.*

*Contacts: e-mail: [support@viva64.com](mailto:support@viva64.com), Tula, 300027, PO box 1800.*

## Lesson 24. Phantom errors

We have finished studying the patterns of 64-bit errors and the last thing we will speak about, concerning

these errors, is in what ways they may occur in programs.

The point is that it is not so easy to show you by an example, as in the following code sample, that the 64-bit code will cause an error when "N" takes large values:

```
size_t N = ...
for (int i = 0; i != N; ++i)
{
    ...
}
```

You may try such a simple sample and see that it works. What matters is the way the optimizing compiler will build the code. It depends upon the size of the loop's body if the code will work or not. In examples it is always small and 64-bit registers may be used for counters. In real programs with large loop bodies an error easily occurs when the compiler saves the value of "i" variable in memory. And now let us make it out what the incomprehensible text you have just read means.

When describing the errors, we often used the term "a potential error" or the phrase "an error may occur". In general, it is explained by the fact that one and the same code may be considered both correct and incorrect depending upon its purpose. Here is a simple example - using a variable of "int" type to index array items. If we address an array of graphics windows with this variable, everything is okay. We do not need to, or, rather, simply cannot work with billions of windows. But when we use a variable of "int" type to index array items in 64-bit mathematical programs or databases, we may encounter troubles when the number of the items exceeds the range 0..INT\_MAX.

But there is one more, subtler, reason for calling the errors "potential": whether an error reveals itself or not depends not only upon the input data but the mood of the compiler's optimizer. Most of the errors we have considered in our lessons easily reveal themselves in debug-versions and remain "potential" in release-versions. But not every program built in the debug mode can be debugged at large data amounts. There might be a case when the debug-version is tested only at small data sets while the exhaustive testing and final user testing at real data are performed in the release-version where the errors may stay hidden.

We encountered the specifics of optimizing Visual C++ 2005 compiler for the first time when preparing the program PortSample. This is a project included into the [PVS-Studio](#) distribution kit which is intended for demonstrating all the errors diagnosed by Viva64 analyzer. We will speak about the project PortSample in more detail in the next lesson. The samples included into this project must work correctly in the 32-bit mode and cause errors in the 64-bit mode. Everything was alright in the debug-version but the release-version caused some troubles. The code that must have hung or led to a crash in the 64-bit mode worked! The reason lay in optimization. The way out was found in excessive complication of the samples' codes with additional constructs and adding the key words "volatile" that you may see in the code of the project PortSample.

The same is with Visual C++ 2008/2010. Of course the code will be a bit different but everything that we will write here may be applied both to Visual C++ 2005 and Visual C++ 2008/2010.

If you find it quite good when some errors do not reveal themselves, put this idea out of your head. Code with such errors becomes very unstable. Any subtle change not even related to the error directly may cause changes in the program behavior. I want to point it out just in case that it is not the compiler's fault - the reason is in the hidden code defects. Further we will show you some samples with phantom errors that disappear and appear again with subtle code changes in release-versions and hunt for which might be

very long and tiresome.

Consider the first code sample that works in the release-version although it must not:

```
int index = 0;
size_t arraySize = ...;
for (size_t i = 0; i != arraySize; i++)
    array[index++] = BYTE(i);
```

This code correctly fills the whole array with values even if the array's size is much larger than INT\_MAX. It is impossible theoretically because the variable *index* has "int" type. Some time later an overflow must lead to accessing the items by a negative index. But optimization gives us the following code:

```
0000000140001040  mov     byte ptr [rcx+rax],cl
0000000140001043  add     rcx,1
0000000140001047  cmp     rcx,rbx
000000014000104A  jne     wmain+40h (140001040h)
```

As you may see, 64-bit registers are used and there is no overflow. But let us make a slightest alteration of the code:

```
int index = 0;
size_t arraySize = ...;
for (size_t i = 0; i != arraySize; i++)
{
    array[index] = BYTE(index);
    ++index;
}
```

Suppose the code looks nicer this way. I think you will agree that it remains the same from the viewpoint of the functionality. But the result will be quite different - a program crash. Consider the code generated by the compiler:

```
0000000140001040  movsxd   rcx,r8d
0000000140001043  mov     byte ptr [rcx+rbx],r8b
0000000140001047  add     r8d,1
000000014000104B  sub     rax,1
000000014000104F  jne     wmain+40h (140001040h)
```

It is that very overflow that must have been in the previous example. The value of the register *r8d* = *0x80000000* is extended in *rcx* as *0xffffffff80000000*. The result is the writing outside the array.

Here is another example of optimization and how easy it is to spoil everything:

```
unsigned index = 0;
for (size_t i = 0; i != arraySize; ++i) {
    array[index++] = 1;
    if (array[i] != 1) {
        printf("Error\n");
        break;
    }
}
```



This is the assembler code:

```

00000000140001040  mov     byte ptr [rdx],1
00000000140001043  add     rdx,1
00000000140001047  cmp     byte ptr [rcx+rax],1
0000000014000104B  jne     wmain+58h (140001058h)
0000000014000104D  add     rcx,1
00000000140001051  cmp     rcx,rdi
00000000140001054  jne     wmain+40h (140001040h)

```

The compiler has decided to use the 64-bit register *rdx* to store the variable *index*. As a result, the code can correctly process an array with a size more than `UINT_MAX`.

But the peace is fragile. Just make the code a bit more complex and it will become incorrect:

```

volatile unsigned volatileVar = 1;
...
unsigned index = 0;
for (size_t i = 0; i != arraySize; ++i) {
    array[index] = 1;
    index += volatileVar;
    if (array[i] != 1) {
        printf("Error\n");
        break;
    }
}

```

The result of using the expression "index += volatileVar;" instead of "index++" is that 32-bit registers start participating in the code and cause the overflows:

```

00000000140001040  mov     ecx,r8d
00000000140001043  add     r8d,dword ptr [volatileVar (140003020h)]
0000000014000104A  mov     byte ptr [rcx+rax],1
0000000014000104E  cmp     byte ptr [rdx+rax],1
00000000140001052  jne     wmain+5Fh (14000105Fh)
00000000140001054  add     rdx,1
00000000140001058  cmp     rdx,rdi
0000000014000105B  jne     wmain+40h (140001040h)

```

In the end let us consider an interesting but large example. Unfortunately, we cannot make it shorter because we need to preserve the necessary behavior to show you. It is the impossibility to predict what a slight change in the code might lead to why these errors are especially dangerous.

```

ptrdiff_t UnsafeCalcIndex(int x, int y, int width) {
    int result = x + y * width;
    return result;
}

...
int domainWidth = 50000;
int domainHeght = 50000;

for (int x = 0; x != domainWidth; ++x)
    for (int y = 0; y != domainHeght; ++y)
        array[UnsafeCalcIndex(x, y, domainWidth)] = 1;

```

This code cannot fill the array consisting of 50000\*50000 items correctly. It cannot do so because an overflow must occur when calculating the expression "int result = x + y \* width;".

Thanks to a miracle, the array is filled correctly in the release-version. The function *UnsafeCalcIndex* is integrated into the loop where 64-bit registers are used:

```

0000000140001052  test     rsi,rsi
0000000140001055  je      wmain+6Ch (14000106Ch)
0000000140001057  lea     rcx,[r9+rax]
000000014000105B  mov     rdx,rsi
000000014000105E  xchg    ax,ax
0000000140001060  mov     byte ptr [rcx],1
0000000140001063  add     rcx,rbx
0000000140001066  sub     rdx,1
000000014000106A  jne     wmain+60h (140001060h)
000000014000106C  add     r9,1
0000000140001070  cmp     r9,rbx
0000000140001073  jne     wmain+52h (140001052h)

```

All this happened because the function *UnsafeCalcIndex* is simple and can be easily integrated. But when you make it a bit more complex or the compiler supposes that it should not be integrated, an error will occur that will reveal itself at large data amounts.

Let us modify (complicate) the function *UnsafeCalcIndex* a bit. Note that the function's logic has not been changed in the least:

```

ptrdiff_t UnsafeCalcIndex(int x, int y, int width) {
    int result = 0;
    if (width != 0)
        result = y * width;
    return result + x;
}

```

The result is a crash, when an access outside the array is performed:

```

0000000140001050  test     esi,esi
0000000140001052  je      wmain+7Ah (14000107Ah)
0000000140001054  mov     r8d,ecx
0000000140001057  mov     r9d,esi
000000014000105A  xchg    ax,ax
000000014000105D  xchg    ax,ax
0000000140001060  mov     eax,ecx
0000000140001062  test     ebx,ebx
0000000140001064  cmovne  eax,r8d
0000000140001068  add     r8d,ebx
000000014000106B  cdq     r8d
000000014000106D  add     rax,rdx
0000000140001070  sub     r9,1
0000000140001074  mov     byte ptr [rax+rdi],1
0000000140001078  jne     wmain+60h (140001060h)
000000014000107A  add     rdx,1
000000014000107E  cmp     rdx,r12
0000000140001081  jne     wmain+50h (140001050h)

```

I hope we have managed to show you how a 64-bit program that works might easily stop doing that after

adding harmless corrections into it or building it with a different version of the compiler.

You will also understand some strange things and peculiarities of the code in PortSample project which are made specially to demonstrate an error in simple examples even in the code optimization mode.

The course authors: Andrey Karpov ([karpov@viva64.com](mailto:karpov@viva64.com)), Evgeniy Ryzhkov ([evg@viva64.com](mailto:evg@viva64.com)).

The rightholder of the course "Lessons on development of 64-bit C/C++ applications" is OOO "Program Verification Systems". The company develops software in the sphere of source program code analysis. The company's site: <http://www.viva64.com>.

Contacts: e-mail: [support@viva64.com](mailto:support@viva64.com), Tula, 300027, PO box 1800.

## Lesson 25. Working with patterns of 64-bit errors in practice

We have finished studying 64-bit error patterns and perhaps you would like to experiment a bit with unsafe constructs and try [PVS-Studio](#) to detect them. You may fulfill your wish by installing the demo project PortSample included into PVS-Studio (Figure 1).

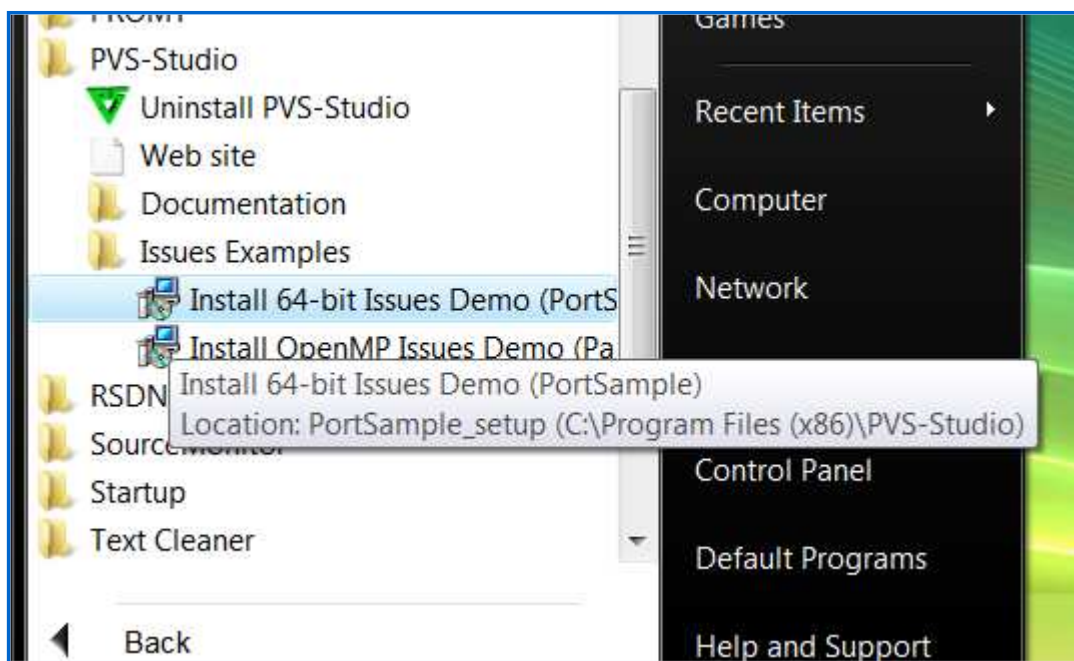


Figure 1 - Installing PortSample project included into PVS-Studio

PortSample is a common C++ project that may be opened both in Visual Studio 2005 and Visual Studio 2008 (Figure 2).

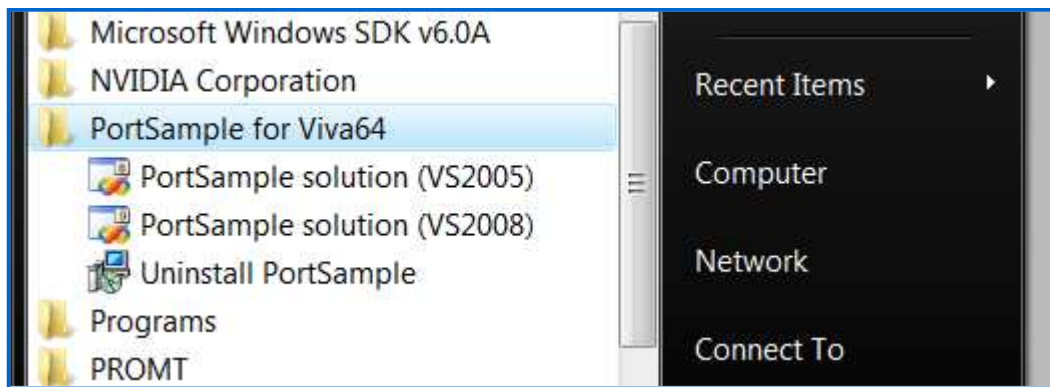


Figure 2 - You may use both Visual Studio 2005 and Visual Studio 2008 to study PortSample

The project PortSample has a collection of examples for all the diagnostic warnings generated by the static analyzer Viva64 included into PVS-Studio when testing 64-bit projects. The warning messages are rather few (at the moment of writing this text, there are 25 messages), but each warning message covers a range of incorrect constructs. It allows us to arrange all the 64-bit errors into common groups with common descriptions. It enables us to avoid repeating the description of one and the same kind of errors in various forms for many times in the documentation. So you can not only look through the [PVS-Studio documentation](#) but really read it to learn all the specifics of 64-bit software coding.

PortSample program's interface shown in Figure 3 makes it convenient to launch the code fragments you are interested in. We do not find it necessary to describe the content of the project in detail - you may study the code you need by yourself. You may also read the article "[Some examples of the 64-bit code errors](#)" where 64-bit errors are thoroughly described by an example from PortSample project.

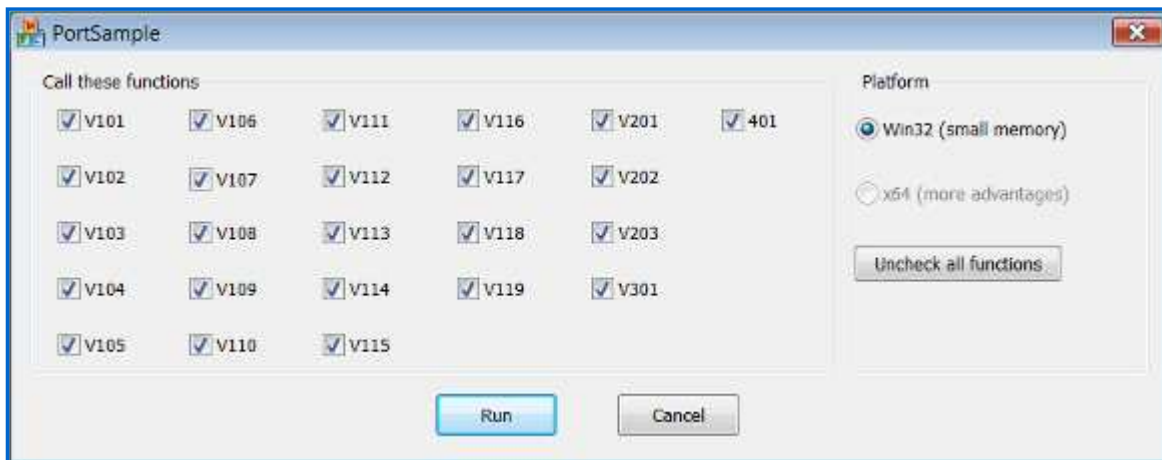


Figure 3 - The program's interface

Note one very important thing concerning the use of the demo version of PVS-Studio. When the demo version is used to check the code, it detects all the potential errors but shows the exact location in the code of only some of them. Instead of the line number you see the text "TRIAL RESTRICTION" as shown in Figure 4.

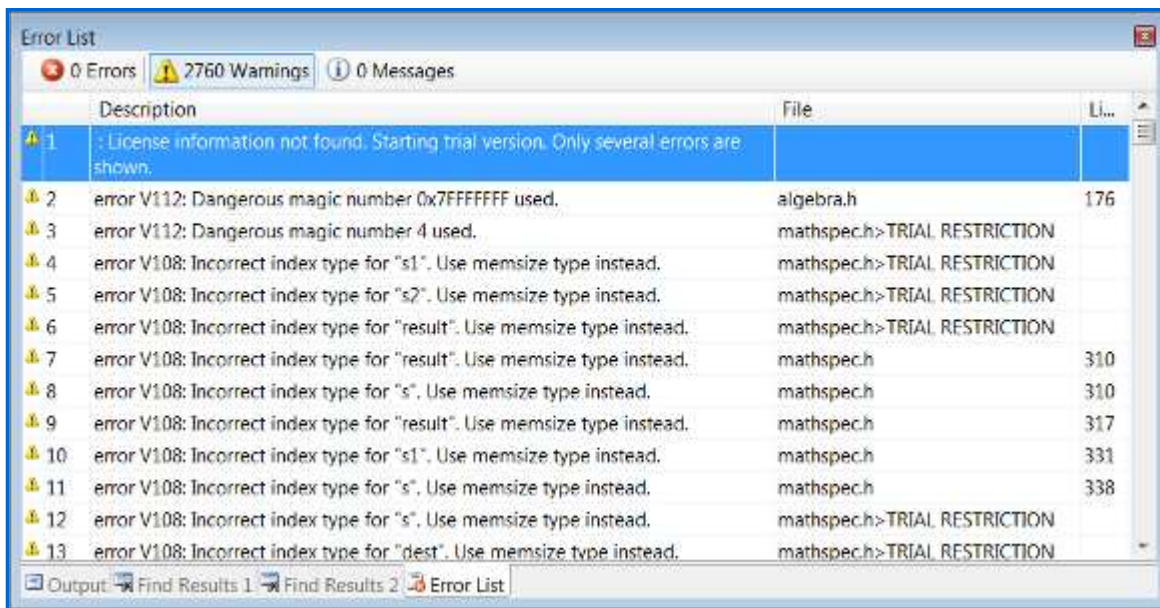


Figure 4 - The demo version of PVS-Studio does not allow you to see all the errors in the code when checking a 64-bit project

When the demo version of PVS-Studio is run with the project PortSample, it shows the location of all the errors. That is, PVS-Studio does not have any restrictions when dealing with samples (see Figure 5). You may modify the PortSample files with errors as you like and thoroughly study PVS-Studio's behavior when processing the code you have written.

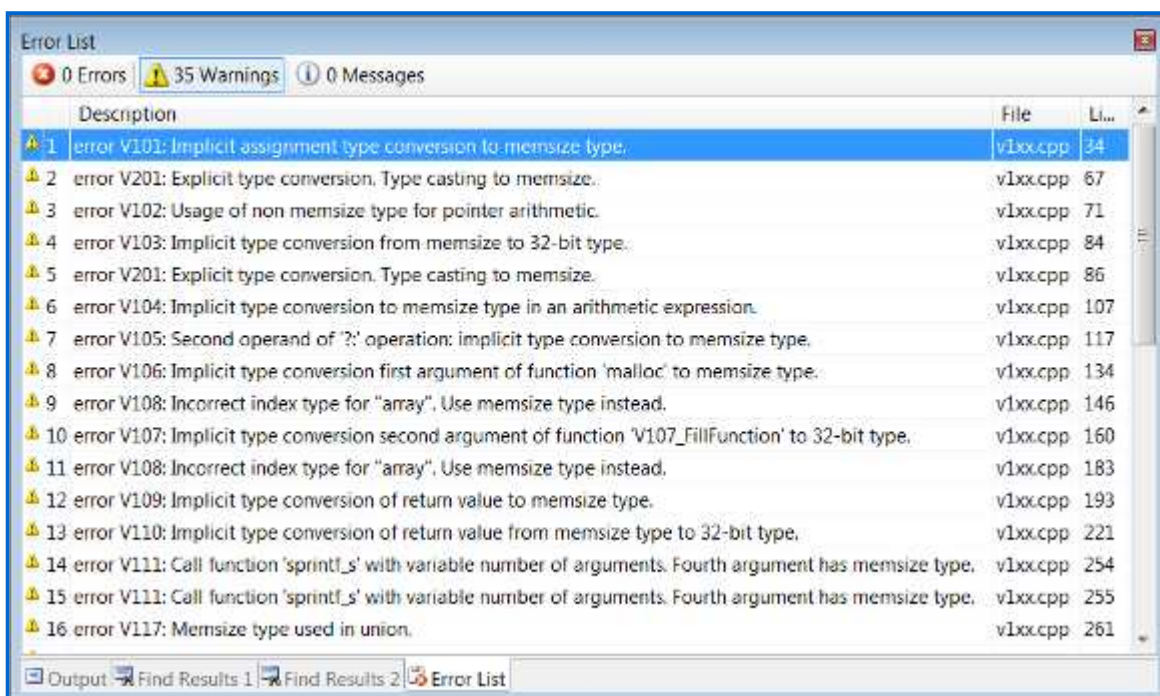


Figure 5 - PVS-Studio demo version shows the numbers of all the lines that contain errors in the project PortSample

If you have questions about using PVS-Studio and PortSample project, ask the tool developers. We will be glad to receive your feedback and recommendations on how to improve PVS-Studio analyzer. Write to us by the address [support@viva64.com](mailto:support@viva64.com).



The course authors: Andrey Karpov ([karpov@viva64.com](mailto:karpov@viva64.com)), Evgeniy Ryzhkov ([evg@viva64.com](mailto:evg@viva64.com)).

The rightholder of the course "Lessons on development of 64-bit C/C++ applications" is OOO "Program Verification Systems". The company develops software in the sphere of source program code analysis. The company's site: <http://www.viva64.com>.

Contacts: e-mail: [support@viva64.com](mailto:support@viva64.com), Tula, 300027, PO box 1800.

## Lesson 26. Optimization of 64-bit programs

### Reducing amounts of memory being consumed

When a program is compiled in the 64-bit mode, it starts consuming more memory than its 32-bit version. This increase often stays unnoticed, but sometimes memory consumption may grow twice. The growth of memory consumption is determined by the following factors:

- larger memory amounts to store some objects, for example pointers;
- changes of the rules of data alignment in structures;
- growth of stack memory consumption.

We can often tolerate the growth of main memory consumption - the advantage of 64-bit systems is that very large amount of memory available to user. It is quite okay if a program takes 300 Mbytes on a 32-bit system with 2 Gbytes of memory and 400 Mbytes on a 64-bit system with 8 Gbytes of memory. In relative units, it means that the program takes three times less memory available on a 64-bit system. So it is unreasonable to strike against the growth of memory consumption we have described - it is easier to add just a bit more memory.

But there is a disadvantage of this growth. It is related to performance loss. Although the 64-bit program code is faster, extraction of larger data amounts from memory might cancel all the advantages and even reduce performance. The operation of transferring data between the memory and microprocessor (cash) is not very cheap.

One of the ways to reduce the memory being consumed is optimization of data structures we have told you about in Lesson 23.

Another way of saving memory is to use more saving data types. For instance, if we need to store a lot of integer numbers and we know that their values will never exceed `UINT_MAX`, we may use the type "unsigned" instead of "size\_t".

### Using memsize-types in address arithmetic

Using `ptrdiff_t` and `size_t` types in address arithmetic might give you an additional performance gain along with making the code safer. For example, using the type `int`, whose size differs from the pointer's capacity, as an index results in additional commands of data conversion appearing in the binary code. We speak about a 64-bit code where the pointers' size is 64 bits while the size of `int` type remains the same - 32 bits.

It is not so easy to give a brief example to show that `size_t` is better than `unsigned`. To be impartial, we have to use the compiler's optimizing capabilities. But two variants of the optimized code often get too

different to easily demonstrate their difference. We managed to create something like a simple example only with a sixth try. But the sample is still far from being ideal because it shows - instead of the unnecessary conversions of data types discussed above - the fact that the compiler can build a more efficient code when using *size\_t*. Consider the program code arranging array items in the reverse order:

```
unsigned arraySize;
...
for (unsigned i = 0; i < arraySize / 2; i++)
{
    float value = array[i];
    array[i] = array[arraySize - i - 1];
    array[arraySize - i - 1] = value;
}
```

The variables "arraySize" and "i" in the example have the type *unsigned*. You can easily replace it with *size\_t* and compare a small fragment of assembler code shown in Figure 1.

array[arraySize - i - 1] = value;	
arraySize, i : <i>unsigned</i>	arraySize, i : <i>size_t</i>
mov eax, DWORD PTR arraySize\$(rsp)	mov rax, QWORD PTR arraySize\$(rsp)
sub eax, r11d	sub rax, r11
add r11d, 1	add r11, 1
<b>add eax, -1</b>	
movss DWORD PTR [rbp+rax*4], xmm0	movss DWORD PTR [rdi+rax*4 <b>-4</b> ], xmm0
...	...

Figure 1 - Comparing the 64-bit assembler code fragments using the types *unsigned* and *size\_t*

The compiler managed to build a more laconic code when using 64-bit registers. We do not want to say that the code created using the type *unsigned* (text on the left) will be slower than the code using the type *size\_t* (text on the right). It is a rather difficult task to compare the speed of code execution on contemporary processors. But you may see from the example that the compiler can build a briefer and faster code when using 64-bit types.

Now let us consider an example showing the advantages of the types *ptrdiff\_t* and *size\_t* from the viewpoint of performance. For the purposes of demonstration, we will take a simple algorithm of calculating the minimum path length. You may see the complete program code here:

<http://www.viva64.com/articles/testspeedexp.zip>.

The function *FindMinPath32* is written in classic 32-bit style with *unsigned* types. The function *FindMinPath64* differs from it only in that way that all the *unsigned* types in it are replaced with *size\_t* types. There are no other differences! I think you will agree that it cannot be considered a complex modification of the program. And now let us compare the execution speeds of these two functions (Table 1).



	Mode and function	Function's execution time
1	32-bit compilation mode. Function FindMinPath32.	1
2	32-bit compilation mode. Function FindMinPath64.	1.002
3	64-bit compilation mode. Function FindMinPath32.	0.93
4	64-bit compilation mode. Function FindMinPath64.	0.85

Table 1 - The time of executing the functions *FindMinPath32* and *FindMinPath64*

Table 1 shows reduced time relative to the speed of execution of the function *FindMinPath32* on a 32-bit system. It was done for the purposes of clearness.

The operation time of the function *FindMinPath32* in the first line is 1 on a 32-bit system. It is explained by the fact that we took this time as a unit of measurement.

In the second line, we see that the operation time of the function *FindMinPath64* is also 1 on a 32-bit system. No wonder, because the type *unsigned* coincides with the type *size\_t* on a 32-bit system, and there is no difference between the functions *FindMinPath32* and *FindMinPath64*. A small deviation (1.002) only indicates a small error in measurements.

In the third line, we see a performance gain of 7%. We could well expect this result after recompiling the code for a 64-bit system.

The fourth line is of the most interest for us. The performance gain is 15%. It means that by merely using the type *size\_t* instead of *unsigned* we let the compiler build a more effective code that works even 8% faster!

It is a simple and obvious example of how data that are not equal to the size of the machine word slow down algorithm performance. Mere replacement of the types *int* and *unsigned* with *ptrdiff\_t* and *size\_t* may result in a significant performance gain. It concerns first of all those cases when these data types are used to index arrays, in address arithmetic and to arrange loops.

*Note. Although the static analyzer [PVS-Studio](#) is not specially designed to optimize programs, it may assist you in code refactoring and therefore make the code more efficient. For example, you will use [memsize-types](#) when fixing potential errors related to address arithmetic, and therefore allow the compiler to build a more optimized code.*

## Intrinsic-function

Intrinsic-functions are special system-dependent functions that perform those actions which cannot be performed at the level of C/C++ code or that perform these functions much more effectively. Actually, they let you get rid of inline-assembler because it is often undesirable or impossible to use it.

Programs may use intrinsic-functions to create faster code due to absence of overhead expenses on calling common functions. The code size will be a bit larger of course. MSDN gives a list of functions

that can be replaced with their intrinsic-versions. For example, these are *memcpy*, *strcmp*, etc.

The compiler Microsoft Visual C++ has a special option `/Oi` that lets you automatically replace the calls of some functions with their intrinsic-analogs.

Besides automatic replacement of common functions with their intrinsic-versions, you may use intrinsic-functions explicitly in your code. This might be helpful due to these factors:

- Inline assembler is not supported by the compiler Visual C++ in the 64-bit mode while intrinsic-code is.
- Intrinsic-functions are simpler to use as they do not require knowledge of registers and other similar low-level constructs.
- Intrinsic-functions are updated in compilers while assembler code must be updated manually.
- The built-in optimizer does not work with assembler code.
- Intrinsic-code is easier to port than assembler code.

Using intrinsic-functions in automatic mode (with the help of the compiler switch) will let you get some free percent of performance gain, and "manual" use even more. That is why using intrinsic-functions is quite reasonable.

To know more about using intrinsic-functions, see the [Visual C++ team's blog](#).

## Alignment

It is good in some cases to help the compiler by defining the alignment manually to enhance performance. For example, SSE data must be aligned on a 16-byte boundary. You may do this in the following way:

```
// 16-byte aligned data
__declspec(align(16)) double init_val[2] = {3.14, 3.14};
// SSE2 movapd instruction
__m128d vector_var = __mm_load_pd(init_val);
```

The sources "[Porting and Optimizing Multimedia Codecs for AMD64 architecture on Microsoft Windows](#)", "[Porting and Optimizing Applications on 64-bit Windows for AMD64 Architecture](#)" cover these issues very thoroughly.

## Other means of performance enhancement

To learn more about the issues of optimizing 64-bit applications, see the document "[Software Optimization Guide for AMD64 Processors](#)".

*The course authors: Andrey Karpov ([karpov@viva64.com](mailto:karpov@viva64.com)), Evgeniy Ryzhkov ([evg@viva64.com](mailto:evg@viva64.com)).*

*The rightholder of the course "Lessons on development of 64-bit C/C++ applications" is OOO "Program Verification Systems". The company develops software in the sphere of source program code analysis. The company's site: <http://www.viva64.com>.*

*Contacts: e-mail: [support@viva64.com](mailto:support@viva64.com), Tula, 300027, PO box 1800.*

## Lesson 27. Peculiarities of creating installers for a 64-bit environment

When developing the 64-bit version of an application, you should also be very attentive to the issue of program distribution - you might encounter some peculiar problems when installing the program on a 64-bit operating system, and if you forget about them, you will get a non-working installation package.

First of all you should understand that the program installer itself (the exe-file that launches the installation process) can technically be either a 32-bit application or a 64-bit one. If you make this installer 64-bit, it will not work on a 32-bit system. Note that you will not see a message like: "You are trying to install a distribution kit of a 64-bit program on a 32-bit system". It will simply generate a message about a damaged file. Thus, it is most often reasonable to make the installer a 32-bit application even if it will be installed only on a 64-bit system.

An important issue of operation of 32-bit programs in the 64-bit environment is the redirection mechanism implemented in Windows. This mechanism arranges the work of obsolete 32-bit applications so that when trying to access, for instance, the folder "c:\program files", they are automatically redirected to "c:\program files (x86)". The access to some register sections is also automatically redirected.

When developing installers, the redirection mechanism may often lead to a situation when the files appear in some other place than the developer has expected, or the register entries relate to some other sections than they should. You may read about the redirection system in MSDN Help system: "[File System Redirector](#)".

All these specifics can be set in any contemporary installer but you should not forget about it while creating a distribution kit compatible with 64-bit operating systems.

When developing installers for 64-bit versions, developers often make one common installer that contains both the 32-bit and 64-bit versions of the applications and their components. In this case, do not forget to add full sets of components and dependent libraries. For example, developers often add Visual C++ Redistributable Package into the installation package for applications developed with Visual C++. This package must be included both for the [x86](#) and [x64](#) versions.

If your installer uses some third-party units to implement complex functionality, you should remember about the restriction - units built for the 64-bit mode cannot load 32-bit dynamic libraries and vice versa.

*The course authors: Andrey Karpov ([karpov@viva64.com](mailto:karpov@viva64.com)), Evgeniy Ryzhkov ([evg@viva64.com](mailto:evg@viva64.com)).*

*The rightholder of the course "Lessons on development of 64-bit C/C++ applications" is OOO "Program Verification Systems". The company develops software in the sphere of source program code analysis. The company's site: <http://www.viva64.com>.*

*Contacts: e-mail: [support@viva64.com](mailto:support@viva64.com), Tula, 300027, PO box 1800.*

## Lesson 28. Estimating the cost of 64-bit migration of C/C++ applications

Starting to plan the port of your project to a 64-bit system, you must be able to estimate the amount of work and material costs. Let us consider those components that make up the total cost of creating a 64-bit software product.

If you find it difficult to estimate the cost of the move to 64-bit systems, you may contact experts of our company "[Program verification systems](#)" for advice. Our company can also take a part of or all the work of adapting your projects for 64-bit systems.

## **Purchasing 64-bit hardware and software**

Nowadays you can hardly find a developer who has a computer with a 32-bit microprocessor. But you still should remember that you must have a 64-bit computer that will let you manage all the tasks you have to solve. It seems that the most real case is when 64-bit hardware architectures still work under 32-bit operating systems. You should take into consideration expenses on purchasing and installing 64-bit versions of operating systems. Consider also additional expenses related to changing the version of the operating system, for example, reinstallation of various software.

## **Purchasing a compiler to build 64-bit applications**

Add the price of purchasing and mastering new compilers able to create 64-bit code to the total price.

## **Purchasing 64-bit versions of libraries**

You might need to purchase 64-bit versions of libraries and other components. Find out beforehand about the pricing policy of those companies whose components are used in your project. Sometimes 32-bit and 64-bit versions of components are sold separately. If you use open-source libraries that have no 64-bit configurations yet, be ready to spend much time on modifying them manually.

## **Staff training and purchasing additional tools**

Take into consideration the time needed for your employees to study all the necessary information on 64-bit system development. You may also need to buy some additional tools such as, for example, [PVS-Studio](#).

## **Code modification**

As you already know from the previous lessons, compiling a 64-bit configuration is only the beginning. In most cases, you will need to find and correct a lot of defects that will occur in 64-bit code. It is perhaps the most laborious yet most difficult to estimate part of the work. However we can advise you the following way relying on PVS-Studio static analyzer.

Well, you have several (tens, hundreds) Mbytes of source code ready for migration. There is no 64-bit configuration of the code yet. So, there are no files to be compiled in the 64-bit mode as well.

PVS-Studio provides you with a capability to detect 64-bit code issues even in 32-bit projects. It is this capability that will let you estimate the difficulty of migration BEFORE creating the 64-bit configuration of your project.

I would like the readers to note how the check of the code is performed in the 32-bit mode. You should understand that this check cannot be considered complete and even if you correct all the errors detected, you cannot be absolutely sure that the code will work in the 64-bit mode. Code of any serious application has such fragments:

```
#ifdef WIN64
...
#endif
```

Of course, this fragment will be skipped when testing the code in the 32-bit mode. Or, more exactly, while there is no 64-bit configuration yet, the application code might have no such a fragment.

Here is another important thing: it is natural that data types differ depending upon the project configuration. That is why the check in the 32-bit and 64-bit modes will nearly always lead to different results.

But how much different will they be? According to the results of the experiments carried out in our company, we have the following: the lists of diagnostic warnings generated by PVS-Studio analyzer when testing projects in the 32-bit and 64-bit modes coincide 95-97%. It means that not more than 5% of diagnostic messages differ.

These results were obtained in the following way. We took the code of real projects, checked it in the 32-bit mode and saved the list of diagnostic warnings. Then we checked the code of the same projects in the 64-bit mode and also saved the list of diagnostic warnings. After that we compared these lists and estimated how many percent of the diagnostic messages coincided. Since the whole procedure was performed in automatic mode, the number of projects that were analyzed was enough (more than 20 projects with the code size of several Mbytes each). So we may conclude that the figures (5% difference) can be trusted.

Of course, you should not hurry to fix all the potential errors detected in the 32-bit configuration of your project – it is better to wait for the 64-bit configuration. But you can easily estimate how much time you will need to check all the warnings generated by the code analyzer.

We recommend you to do the following:

1. Analyze the 32-bit configuration of the project with PVS-Studio.
2. One programmer who knows well the issues of 64-bit code looks through the warnings generated by the analyzer during a day and decides if this or that error is relevant to the project. If it is, the programmer corrects it.
3. The total number of the analyzer-generated messages is divided by the number of the messages the programmer has looked through and processed during one day.
4. The number you get is the number of man-days needed to port the application's code to a 64-bit platform.

The programmer must correct the errors found. It is not enough just to find an error and imagine that it is corrected. To detect and to correct are actions that differ in time they take. You may need to modify the program code in many project files to correct some errors. To avoid an understated estimate, you must make all the necessary corrections.

Of course, there is a drawback in this algorithm of estimating the migration process – it is the skill of the developer who will process the messages of the analyzer and modify the code during a day. So we

recommend you to be very serious and careful when choosing a programmer responsible for the estimate.

Here are some recommendations on how to choose such a programmer:

1. This person must be an experienced programmer who has been working in your company not less than for three years and who knows this particular project you want to port.
2. The programmer must be familiar with the issues of 64-bit code – for example, know these lessons or the article "[20 issues of porting C++ code on the 64-bit platform](#)".
3. It is desirable that the programmer understand the principles of working with static code analyzers. It is not an obligatory requirement but understanding the static code analysis technology makes the estimate of the migration process more adequate.
4. The person must be able to stay within the usual working conditions during the testing day. He or she must not try to set up a record of performance to impress the colleagues. One cannot work all the days in such a way, and the terms will be estimated incorrectly.

Following these recommendations will allow you to get an adequate estimate of the cost and term of the 64-bit software migration process.

## Adapting the testing system

Consider the cost of adapting your testing system for full-fledged testing of 64-bit units. If your programs process large data amounts, you must have tests that run on data amounts more than 4 Gbytes. In its turn, integration of heavy tests might result in the task of test parallelizing. In this case you might have to buy additional tools.

## Protection of software units

If you use software or firmware systems of software copying and cracking protection, you should add the price of implementing protection for your 64-bit code to the total cost. Perhaps you will have to master new protection systems if those systems you are using at present do not support 64-bit codes. You might face other unexpected troubles, so make sure you have some time in reserve to manage them.

## Distribution kit adaptation

You will have to create a new distribution kit – this issue was considered in the previous lesson.

*The course authors: Andrey Karpov ([karpov@viva64.com](mailto:karpov@viva64.com)), Evgeniy Ryzhkov ([evg@viva64.com](mailto:evg@viva64.com)).*

*The rightholder of the course "Lessons on development of 64-bit C/C++ applications" is OOO "Program Verification Systems". The company develops software in the sphere of source program code analysis.*

*The company's site: <http://www.viva64.com>.*

*Contacts: e-mail: [support@viva64.com](mailto:support@viva64.com), Tula, 300027, PO box 1800.*

---

[Home](#) | [About Us](#) | [Services](#) | [Contact Us](#) | [Privacy Policy](#) | [Terms of Use](#) | [Sitemap](#)

© 2008 - 2010, OOO "Program Verification Systems"

300027, Russia, Tula, P.O. Box 1800. Office: Russia, Tula, Kutuzova 100-73

